



Feathers

A minimalist real-time framework for
tomorrow's apps.

The official guide

Table of Contents

Introduction	1.1
Guides	1.2
About Feathers	1.2.1
Features	1.2.1.1
Philosophy	1.2.1.2
Feathers vs. X	1.2.1.3
The Basics	1.2.2
Introduction	1.2.2.1
What not to worry about	1.2.2.2
Installing the Examples	1.2.2.3
Basic Feathers	1.2.2.4
A database connector	1.2.2.5
A REST API server	1.2.2.6
A Feathers REST client	1.2.2.7
A Feathers WebSocket client	1.2.2.8
The "a-ha!" moment	1.2.2.9
Hooks middleware	1.2.2.10
All about hooks	1.2.2.11
Hooks, part 1	1.2.2.12
Hooks, part 2	1.2.2.13
Writing your own hooks	1.2.2.14
Testing hooks	1.2.2.15
Real-time	1.2.2.16
The Generator (CLI)	1.2.3
Generate the application	1.2.3.1
Add authentication	1.2.3.2
Add the teams service	1.2.3.3
Add the populate hook	1.2.3.4
Run the application	1.2.3.5
A Chat Application	1.2.4
Creating the application	1.2.4.1
Generating a service	1.2.4.2
Building a frontend	1.2.4.3
Adding authentication	1.2.4.4
Processing data	1.2.4.5
Frameworks	1.2.5
React and React Native	1.2.5.1

VueJS	1.2.5.2
Angular	1.2.5.3
Others	1.2.5.4
Authentication	1.2.6
How JWT works	1.2.6.1
What's New (external link)	1.2.6.2
Migration Guide (external link)	1.2.6.3
Recipe: Custom Login Response	1.2.6.4
Recipe: Custom JWT Payload	1.2.6.5
Recipe: Mixed Auth Endpoints	1.2.6.6
Recipe: Basic OAuth	1.2.6.7
Offline first	1.2.7
Strategies	1.2.7.1
Snapshot	1.2.7.2
Realtime	1.2.7.3
Optimistic mutation	1.2.7.4
Own-data, own-net	1.2.7.5
Sync-data, sync-net	1.2.7.6
Configure snapshot	1.2.7.7
Configure realtime	1.2.7.8
Configure publication	1.2.7.9
Example snapshot	1.2.7.10
Example realtime & publication	1.2.7.11
Example optimistic mutation	1.2.7.12
Tests as examples	1.2.7.13
More examples	1.2.7.14
Advanced topics	1.2.8
Debugging	1.2.8.1
Configuration	1.2.8.2
File uploads	1.2.8.3
Creating a Feathers plugin	1.2.8.4
Seeding services	1.2.8.5
Using a view engine	1.2.8.6
Scaling	1.2.8.7
API	1.3
Core	1.3.1
Application	1.3.1.1
Services	1.3.1.2
Hooks	1.3.1.3
Common Hooks	1.3.1.4

Client	1.3.1.5
Events	1.3.1.6
Errors	1.3.1.7
Transports	1.3.2
 REST	1.3.2.1
 Express	1.3.2.2
 Socket.io	1.3.2.3
 Primus	1.3.2.4
Authentication	1.3.3
 Server	1.3.3.1
 Client	1.3.3.2
 Local	1.3.3.3
 Local management	1.3.3.4
 JWT	1.3.3.5
 OAuth1	1.3.3.6
 OAuth2	1.3.3.7
 Hooks	1.3.3.8
Databases	1.3.4
 Common API	1.3.4.1
 Querying	1.3.4.2
 Memory	1.3.4.3
 NeDb	1.3.4.4
 LocalStorage	1.3.4.5
 MongoDB	1.3.4.6
 Mongoose	1.3.4.7
 Sequelize	1.3.4.8
 Knex	1.3.4.9
 RethinkDB	1.3.4.10
 Elasticsearch	1.3.4.11
Security	1.4
Ecosystem	1.5
Help	1.6
FAQ	1.7
Contributing	1.8
License	1.9



Feathers

An open source REST and realtime API layer for modern applications.

With Feathers it's easy to create scalable, real-time applications. Make creating web and mobile apps fun with Feathers.

This documentation is also available as a [PDF](#).

[Guides](#)

Get familiar with Feathers by building your first apps. Learn topics from beginner to more advanced-levels.

[API](#)

Learn more about Feathers' Universal API, and plugins for Authentication and handling data.

[Security](#)

Understand our commitment to security and how it affects your applications.

[Ecosystem](#)

See what amazing things the Feathers Core Team and Community have built.

[Help](#)

Learn how to plug in to the active and helpful FeathersJS Community.

[FAQ](#)

A collection of Frequently Asked Questions.

[Contributing](#)

Learn how you can contribute to this documentation.

[License](#)

Guides

Official guides

About Feathers

Learn about Feathers features, philosophy and how it compares with some other frameworks.

The Basics - A Step-by-Step Intro to Feathers

The goal of this guide is to get you to the "A-ha!" moment as efficiently as possible. You will learn how the primary parts of the core work together. You'll also learn how to start new applications with the generator.

The Generator (CLI)

An overview of the `feathers-cli` and the application it generates.

A Chat Application

Learn how to create a chat REST and real-time API complete with authentication and data processing and how to use Feathers on the client in a simple browser app.

Authentication

Learn how to add local (username & password), OAuth1, and OAuth2 authentication to your Feathers Applications.

Advanced topics

Guides for more advanced Feathers topics like debugging, configuration, file uploads and more.

Video tutorials

The FeathersJS Youtube playlist



Feathers

A minimalist real-time framework for tomorrow's apps.



A growing collection of Feathers related talks, tutorials and discussions.

FeathersJS Real-Time Chat App - Tutorial

```

<script>
  </ul>
  </div>
</template>

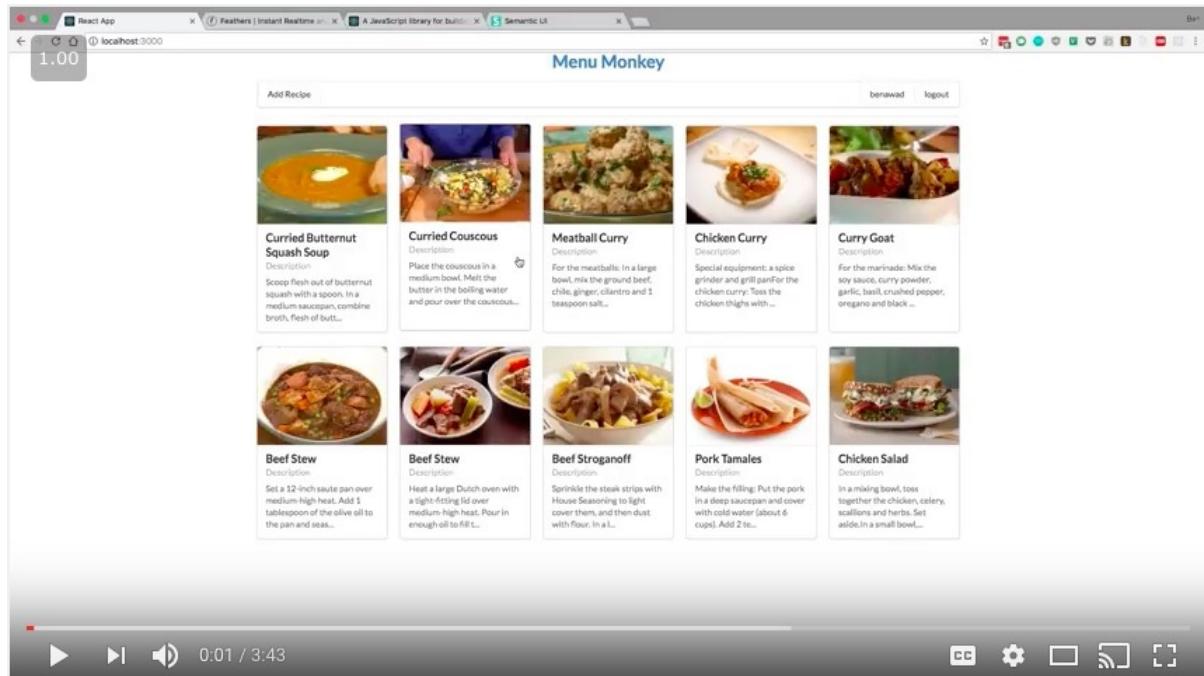
<script>
  import * as services from '../services'
  import { getMessages } from '../vuex/getters'
  import { fetchMessages, addMessage, removeMessage } from '../vuex/actions'

  export default {
    vuex: {
      getters: {
        messages: getMessages
      },
      actions: {
        fetchMessages,
        addMessage,
        removeMessage
      }
    },
    data () {
      return {
        newMessage: ''
      }
    },
    ready () {

```

Mad ❤ to [Chris Pena](#) for putting together the video.

Fullstack Feathersjs and React Web App



Mad ❤ to [Ben Awad](#) for putting together an entire video series.

Why Feathers

Using Feathers is a great way to build scalable, real-time web and mobile applications. You can literally build prototypes in minutes and production ready applications in days. Feathers achieves this by being a thin wrapper over top of some amazing battle tested open source technologies and adding a few core pieces like [Hooks](#) and [Services](#).

If you've decided that Feathers might be for you and you haven't tried the tutorial, feel free to dive right in and [learn about the basics](#). If you're still unsure about what Feathers does and where it comes from see [what Feathers offers](#), learn more about the [Feathers philosophy](#) or check out [how Feathers compares to others](#).

Features

Feathers provides a lot of the things that you need for building modern web and mobile applications. Here are some of the things that you get out of the box with Feathers. All of them are optional so you can choose exactly what you need. No more, no less.

We like to think of Feathers as a "*batteries included but easily swappable*" framework.

Instant REST APIs	Feathers automatically provides REST APIs for all your services. This industry best practice makes it easy for mobile applications, a web front-end and other developers to communicate with your application.
Unparalleled Database Support	With Feathers service adapters you can connect to all of the most popular databases, and query them with a unified interface no matter which one you use. This makes it easy to swap databases and use entirely different DBs in the same app without changing your application code.
Real Time	Feathers services can notify clients when something has been created, updated or removed. To get even better performance, you can communicate with your services through websockets, by sending and receiving data directly.
Cross-Cutting Concerns	Using "hooks" you have an extremely flexible way to share common functionality or concerns . Keeping with the Unix philosophy, these hooks are small functions that do one thing and are easily tested but can be chained to create complex processes.
Universal Usage	Services and hooks are a powerful and flexible way to build full stack applications. In addition to the server, these constructs also work incredibly well on the client. That's why Feathers works the same in NodeJS, the browser and React Native.
Authentication	Almost every app needs authentication so Feathers comes with support for email/password, OAuth and Token (JWT) authentication out of the box.
Pagination	Today's applications are very data rich so most of the time you cannot load all the data for a resource all at once. Therefore, Feathers gives you pagination for every service from the start.
Error Handling	Feathers removes the pain of defining errors and handling them. Feathers services automatically return appropriate errors, including validation errors, and return them to the client in a easily consumable format.

The Feathers Philosophy

We know! You're probably screaming "*Not another JavaScript framework!*". We've also become frustrated with all the Rails clones and MVC frameworks that don't do anything different. Instead, a few years ago we started to explore a different approach to building web applications using services and cross cutting concerns while also being careful not to reinvent the wheel.

With this experimentation Feathers has grown into what it is today. Our core philosophy that guides Feathers is still the same as it was years ago:

"Monolithic apps tend to fall apart at scale, either because of performance or because there are too many people in the code. What if we could make it easy to build applications that can naturally become service oriented from day one, rather than having to start with a large application and painfully tease it apart?"

"What if we could make a framework that grows with you and your business and makes it easy for you to transition to a series of microservices, or easily change databases without ripping our code apart?"

"What if we could make real-time less intimidating rather than a hacky, complex after thought? What if we could remove the boilerplate needed for building REST APIs? Could we build a framework that provides enough structure to get going easily and add all the common pieces that modern apps need, but still keep everything flexible and optional?"

"A framework itself should not be opinionated. It should be made up of small, reusable, optional components that do one thing well but are combined in an opinionated way. By keeping the components of your application small, flexible and optional you eliminate much of the engineering obstacles that prevent moving fast and scaling."

We strongly believe that your UI, data and business logic are the core of any web or mobile application and your framework should take care of the rest so you can focus on the things that matter.

Services

Many web frameworks focus on things like rendering views, defining routes and handling HTTP requests and responses without providing a structure for implementing application logic separate from those secondary concerns. The result - even when using the MVC pattern - are monolithic applications with messy controllers or fat models. Your actual application logic and how your data is accessed are all mixed up together.

Feathers brings 3 important concepts together that help to separate those concerns from how your application works and give you incredible flexibility while still keeping things **DRY**.

A service layer which helps to decouple your application logic from how it is being accessed and represented. Besides also making things a lot easier to test - you just call your service methods instead of having to make fake HTTP requests - this is what allows Feathers to provide the same API through both HTTP REST and websockets. It can even be extended to use any other RPC protocol and you won't have to change any of your services.

Uniform Interfaces

Every Feathers service exposes a uniform interface modeled after REST. Where, just like one of the key constraints of REST, your action context is immediately apparent due to the naming convention. With REST you have the HTTP verbs (GET, POST, PUT, PATCH and DELETE). This translates naturally to a Feathers service object interface:

```
const myService = {
  // GET /path
```

```

  find(params, callback) {},
  // GET /path/<id>
  get(id, params, callback) {},
  // POST /path
  create(data, params, callback) {},
  // PUT /path/<id>
  update(id, data, params, callback) {},
  // PATCH /path/<id>
  patch(id, data, params, callback) {},
  // DELETE /path/<id>
  remove(id, params, callback) {}
}

```

This interface also makes it easier to "hook" into the execution of those methods and emit events when they return which can naturally be used to provide real-time functionality.

Hooks

[Cross cutting concerns](#) are an extremely powerful part of aspect oriented programming. They are a very good fit for web and mobile applications since the majority are primarily CRUD applications with lots of shared functionality. Keeping with the Unix philosophy we believe that small modules that do one thing are better than large complex ones. That's why you can create `before` and `after` hooks and chain them together to create very complex processes while still maintaining modularity and flexibility.

Built on the Shoulders of Giants

Because we utilize some already proven modules, we spend less time re-inventing the wheel, are able to move incredibly fast, and have small well-tested, stable modules.

Here's how we use some of the tech under the hood:

- Feathers extends [Express 4](#), the most popular web framework for [NodeJS](#).
- Our CLI tool uses [commander](#) and its generators are built with [Yeoman](#).
- We wrap [Socket.io](#) or [Primus](#) as your websocket transport.
- Our service adapters typically wrap mature ORMs like [mongoose](#), [sequelize](#) or [knex](#).
- [npm](#) for package management.
- [passport](#) for much of the [feathers-authentication](#) work.

Feathers vs X

The following sections compare Feathers to other software choices that seem similar or may overlap with the use cases of Feathers.

Due to the bias of these comparisons being on the Feathers website, we attempt to only use facts. Below you can find a feature comparison table and in each section you can get more detailed comparisons.

If you find something invalid or out of date in the comparisons, please [create an issue](#) (or better yet, a [pull request](#)) and we'll address it as soon as possible.

Feature Comparison

Due to the fact that ease of implementation is subjective and primarily related to a developer's skill-set and experience we only consider a feature supported if it is officially supported by the framework or platform, regardless of how easy it is to implement (aka. are there official plugins, guides or SDKs?).

Legend

- : Officially supported with a guide or core module
- : Not supported
- : Community supported or left to developer

Feature	Feathers	Express	Meteor	Sails	Firebase
REST API					
Real Time From Server					
Real Time From Client			(DDP)		
Universal JavaScript			(sort of)		
React Native Support					
Client Agnostic					(SDKs)
Email/Password Auth					
Token Auth					
OAuth					
Self Hosted					
Hosting Support					
Pagination					
Databases	10+ databases. Multiple ORMs		MongoDB	10+ databases. 1 ORM	Unknown

Analytics					
Admin Dashboard					
Push Notifications					
Offline Mode					
Hot Code Push					

Feathers vs Firebase

Firebase is a hosted platform for mobile or web applications. Just like Feathers, Firebase provides REST and real-time APIs but also includes CDN support. Feathers on the other hand leaves setting up a CDN and hosting your Feathers app up to the developer.

Firebase is a closed-source, paid hosted service starting at 5\$/month with the next plan level starting at 49\$/month. Feathers is open source and can run on any hosting platform like Heroku, Modulus or on your own servers like Amazon AWS, Microsoft Azure, Digital Ocean and your local machine. Because Firebase can't be run locally you typically need to pay for both a shared development environment on top of any production and testing environment.

Firebase has JavaScript and mobile clients and also provides framework specific bindings. Feathers currently focuses on universal usage in JavaScript environments and does not have any framework specific bindings. Mobile applications can use Feathers REST and websocket endpoints directly but at the moment there are no Feathers specific iOS and Android SDKs.

Firebase currently supports offline mode whereas that is currently left up to the developer with Feathers. We do however have [a proposal](#) for this feature.

Both Firebase and Feathers support email/password, token, and OAuth authentication. Firebase has not publicly disclosed the database technology they use to store your data behind their API but it seems to be an SQL variant. Feathers supports [multiple databases](#), NoSQL and SQL alike.

For more technical details on the difference and how to potentially migrate an application you can read [how to use Feathers as an open source alternative to Firebase](#).

Feathers vs Meteor

Both Feathers and Meteor are open source real-time JavaScript platforms that provide front end and back end support. They both allow clients to send and receive messages over websockets. Feathers lets you choose which real-time transport(s) you want to use via [socket.io](#) or [Primus](#), while Meteor relies on SockJS.

Feathers is community supported, whereas Meteor is venture backed and has raised 31.2 million dollars to date.

Meteor only has official support for MongoDB but there are some community modules of various levels of quality that support other databases. Meteor has its own package manager and package ecosystem. They have their own template engine called Blaze which is based off of Mustache along with their own build system, but also have guides for Angular and React.

Feathers has official support for [many more databases](#) and supports any front-end framework or view engine that you want by working seamlessly [on the client](#).

Feathers uses the defacto JavaScript package manager [npm](#). As a result you can utilize the hundreds of thousands of modules published to npm. Feathers lets you decide whether you want to use Gulp, Grunt, Browserify, Webpack or any other build tool.

Meteor has optimistic UI rendering and oplog tailing whereas currently Feathers leaves that up to the developer. However, we've found that being universal and utilizing websockets for both sending and receiving data alleviates the need for optimistic UI rendering and complex data diffing in most cases.

Both Meteor and Feathers provide support for email/password and OAuth authentication. Once authenticated Meteor uses sessions to maintain a logged in state, whereas Feathers keeps things stateless and uses [JSON Web Tokens \(JWT\)](#) to assess authentication state.

One big distinction is how Feathers and Meteor provide real-time across a cluster of apps. Feathers does it at the service layer or using another pub-sub service like Redis whereas Meteor relies on having access to and monitoring MongoDB operation logs as the central hub for real-time communication.

Feathers vs Sails

From a feature standpoint, Feathers and Sails are probably the most similar of the comparisons offered here. Both provide real-time REST API's, multiple DB support, and are client-agnostic. Sails is bound to the server whereas Feathers can also be used in the browser and in React Native apps. Both frameworks use Express, with Feathers supporting the latest Express 4, while Sails supports Express 3.

Sails follows the MVC pattern while Feathers provides lightweight services to define your resources. Feathers uses hooks to define your business logic including validations, security policies, and serialization in reusable, chainable modules, whereas with Sails, these reside in more of a configuration file format.

Feathers supports multiple ORMs while Sails only supports its own Waterline ORM.

Sails allows you to receive messages via websockets on the client, but, unlike Feathers, does not directly support data being sent from the client to the server over websockets. Additionally, Sails uses Socket.io for its websocket transport. Feathers also supports Socket.io but also many other socket implementations via [Primus](#).

Even though the features are very similar, Feathers achieves this with much less code. Feathers also doesn't assume how you want to manage your assets or that you even have any (because you might be making a JSON API). Instead of coming bundled with Grunt, Feathers lets you use your build tool of choice.

Sails doesn't come with any built-in authentication support. Instead, it offers guides on how to configure Passport. By contrast, Feathers supports an [official authentication plugin](#) that is a drop-in, minimal configuration, module that provides email/password, token, and OAuth authentication much more like Meteor. Using this you can authenticate using those providers over REST **and/or** sockets interchangeably.

Scaling a Sails app is as simple as deploying your large app multiple times behind a load balancer with some pub-sub mechanism like Redis. With Feathers you can do the same but you also have the option to mount sub-apps more like Express, spin up additional services in the same app, or split your services into small standalone microservice applications.

Step by Step Intro to Basic Feathers

Feathers is a REST and realtime API layer for modern applications.

[FeathersJS] signature feature [is] that it's super lightweight. It contains a simple and logical workflow that streamlines building apis and can make an api that would have taken hours and builds it in minutes. It hits the perfect balance of magic and control where you still have full control over how your api behaves but the tools provided make your life so much easier. -- Medium - "FeathersJS—A framework that will spoil you"

Warning: Feathers is addictive.

Services

[Services](#) are the heart of Feathers, as this is what all clients will interact with. They are middlemen and can be used to perform operations of any kind.

- interact with a database
- interact with a microservice/API
- interact with the filesystem
- interact with other resources
 - send an email,
 - process a payment,
 - return the current weather for a location, etc.

Hooks

[Hooks](#) are functions that run automatically before or after a service is called upon. They can be service gatekeepers and make sure that all operations are allowed and have the required information. They can also make sure that only data that should be returned to a client is returned.

- before hooks: validate/cleanse/check permissions.
- after hooks: add additional data or remove unneeded data before it's sent to the client.

Events

[Events](#) are sent to clients (or other servers if the feathers-sync package is used) when a service method completes. The `created`, `updated`, `patched`, and `removed` events provide real-time functionality

[Event Filtering](#) determines which users should receive an event. This is the Feathers alternative to Socket.io's rooms and it's an extremely intelligent approach that enables reactive applications to scale well.

Authentication

Feathers provides [local](#), [JSON Web Token](#), [OAuth1](#) and [OAuth2](#) authentication (using [PassportJS](#)) over [REST](#) and [WebSockets](#).

Providers

Choose which providers to use in your application.

- REST
- Socket.io
- Primus (supporting [engine.io](#), [uWebSockets](#) a.k.a. [uws](#), [SockJS](#), [Faye](#))

Middleware

[Express middleware](#) handles the extra fluff that isn't exactly necessary, but can be nice for optimization/logging.

- before service methods: compression, CORS, etc.
- after service methods: logs, error handlers, etc.

This guide's purpose

This guide covers

- Services used with a database.
- Hooks.
- Events.
- Providers.

It does not assume any prior knowledge of Feathers.

By the time you finish this guide, you will

- have a solid understanding of Feathers basics.
- understand how Feathers permits your code to be database agnostic.
- understand how a Feathers server simultaneously and transparently supports a HTTP REST API, Feathers REST clients, and Feathers WebSocket clients.
- understand that you can access your database from the client as if that client code was running on the server.
- understand that the Feathers generators will structure your application for you, and you will understand the boilerplate they produce.

By the time you finish this guide, you will be ready to write your first app.

Introduction

We will start with writing snippets of Feathers code by hand. We'll take a step by step approach, introducing a few new concepts each time.

Each step is backed by a working example in the `examples/step/` folder of the docs. The code samples in this guide are extracts from those examples. Code snippets may be ambiguous, misleading or confusing. Working examples reduce these problems, and let you learn more by modifying them yourself.

One example may continue with changes from a previous example. In such cases, a summary of the differences between the two examples may be shown to help you understand the changes.

Warning. The clients in the examples log results to the browser console. So open the console log before pointing the browser at an example URL.

Feathers has a definite **a-ha!** moment, that moment when you realize how much it accomplishes and how simply. We want to get to that moment quickly, while fully understanding what is happening.

We'll develop a solid enough understanding of Feathers basics that, by the time we get to Feathers' generators, we'll be mostly interested in how they structure projects rather than in the code they produce.

Our intended audience

Readers should have reasonable JavaScript experience, some experience with [Node](#), the concept of [HTTP REST](#), and an idea of what [WebSockets](#) are. Having some experience with [ExpressJS](#) is an asset. We assume everyone has worked with database tables.

This guide should be a comfortable introduction to Feathers for people learning new technologies, such as those coming from PHP, Ruby, or Meteor.

It may be productive for seasoned developers, experienced in Node, REST and WebSockets, to skim the text, paying more attention to the code extracts.

They should however make sure to absorb fully the [Generators](#) section. That should save them some time compared to putting together their own understanding of how projects are structured.

What not to worry about at this time

How do I structure my app?

The generators will do it for you.

How do I use my preferred database?

Feathers supports over 20 different databases. Feathers apps are database agnostic for the most part. At worst, it shouldn't take more than 30 minutes to switch your app from one database to another.

Authentication.

Feathers authentication wraps [PassportJS](#) so Feathers can do anything Passport does.

How do I use Feathers with React, Angular, Vue?

These are covered in the companion guides.

Is Feathers production ready? Is it scalable?

Yes, and yes. There's detailed information in [this post](#).

Javascript Promises.

Feathers works with both [callbacks](#) or [Promises](#). This guide uses Promises as they are prioritized by the Feathers team. We'll be explaining what you need to know about Promises, when you need to know it.

Is anything wrong, unclear, missing?

[Leave a comment](#).

Installing the examples

You can install the code for the examples used in this guide, which would allow you to run those examples.

This however is not a requirement and we suggest you first just read through the guide. You can always install the examples later if you want to work with them.

Install Node

[Node](#) is a server platform which runs JavaScript. It's lightweight and efficient. It has the largest ecosystem of open source libraries in the world.

- [Default install](#).
- [Specific versions](#).

Install git

[git](#) is the version control system most frequently used in open source. There are many resources available for installing it.

- [Linux](#).
- [macOS](#).
- [Windows](#).

Install the examples

From your terminal:

```
cd the/folder/above/which/you/want/the/guide/to/reside  
git clone https://github.com/feathersjs/feathers-docs
```

Alternative install. If you don't already have git installed on your machine, you may prefer to download the repository as a zip file. Point your browser at <https://github.com/feathersjs/feathers-docs/archive/master.zip> to start the download.

Install dependencies used by the examples

```
cd path/to/feathers-docs/examples/step/01  
npm install
```

This will install the dependencies needed by the Basics examples into `/examples/step/01/node_modules`.

Recreating the examples used in the guides

Each guide is divided into sections, each section backed by working examples in `examples/`. The code samples in the guides are extracts from those examples. Code snippets may be ambiguous, misleading, or confusing. Working examples reduce these problems, as well as let you learn more by modifying them yourself.

One example may continue with changes from a previous example. In such cases, a recap of the differences between the two examples may be shown to help in understanding the changes.

The guides go into details about how each example was created. You can recreate the process yourself if that helps your learning process. Create a folder called, say, `copy-an-introduction` with a subfolder `examples/`. You can run the same commands as mentioned in the guide and (hopefully!) get the same results.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Basic Feathers

| Getting you to Feathers' **a-ha!** moment as quickly as possible.

Let's write some Feathers code because, once we understand that, Feathers generators will be straight forward. We'll start by using a database connector in [the next section](#).

Is anything wrong, unclear, missing?

[Leave a comment.](#)

A Database Connector

Our first Feathers example resides on the server only. We'll see how easy it is to use a database table.

This example adds some user items to a NeDB database table, reads them back and displays them.

Databases. We're using the NeDB database because it won't distract us from concentrating on Feathers.

NeDB resembles the popular MongoDB database but requires neither installation nor configuration.

Feathers supports over 20 different databases. Everything we mention in this guide is applicable to all of them.

Working example

- Source code: [examples/step/01/db-connector/1.js](#)
- Run it: `node ./examples/step/01/db-connector/1.js`

The source code contains lines like `/// [dependencies]` and `//! [dependencies]`. Ignore them. They are for automatic display of code snippets in this guide.

Feathers is modular

Feathers embodies the same spirit as the popular HTTP server framework [Express](#). Feathers is comprised of small modules that are all completely optional, and the core weighs in at just a few hundred lines of code. How's that for light weight! Now you can see where Feathers got its name.

We require our dependencies.

```
const feathers = require('feathers');
const NeDB = require('nedb');
const path = require('path');
const service = require('feathers-nedb');
```

We start an instance of Feathers and define its services.

```
const app = feathers()
  .configure(services);
```

users is the only service we need and it's a database table located at `examples/step/data/users.db`.

```
function services() {
  this.use('/users', service({ Model: userModel() }));
}

function userModel() {
  return new NeDB({
    filename: path.join('examples', 'step', 'data', 'users.db'),
    autoload: true
});
```

```
}
```

Create 3 users using Promises.

```
const users = app.service('/users');

Promise.all([
  users.create({ email: 'jane.doe@gmail.com', password: '11111', role: 'admin' }),
  users.create({ email: 'john.doe@gmail.com', password: '22222', role: 'user' }),
  users.create({ email: 'judy.doe@gmail.com', password: '33333', role: 'user' })
])
```

Each create returns a promise which resolves into the item added into the database. NeDB will always add a unique `_id` property to the user item and the returned item will contain it.

Callbacks and Promises. `users.create({ ... }, {}, (err, data) => { ... })` would create a user item using a callback signature. We however will use Promises in this guide because the Feathers team prioritizes them.

Promise Refresher. `Promise.all([...]).then(results => { ... })`; `Promise.all` takes an array whose elements are JavaScript values or Promises. It returns a single Promise that will resolve if **every** promise in the array is resolved or reject if **any** promise in the array is rejected. The elements are resolved in parallel, not sequentially, so `Promise.all` is a great pattern with which to start independent actions. The `then` portion is called once all elements are resolved. It receives an array as a parameter. The n-th element of the array is the resolved value of the n-th element in `Promise.all`.

The 3 user items are now in the database, their values are returned in `results`. We issue a find for the entire table and print the results.

```
.then(results => {
  console.log('created Jane Doe item\n', results[0]);
  console.log('created John Doe item\n', results[1]);
  console.log('created Judy Doe item\n', results[2]);

  return users.find()
    .then(results => console.log('find all items\n', results));
})
.catch(err => console.log('Error occurred:', err));
```

Promise Refresher. `user.find().then(results => ...)`; `user.find()` returns a Promise. `.then(results => ...)` waits for the Promise to resolve, i.e. for the find to finish. The zero, one or more items found in the table are returned in the `results` param.

| View the completed file [db-connector/1.js](#).

Service methods

Feathers provides the following [service methods](#):

```
find(params)
get(id, params)
create(data, params)
update(id, data, params)
patch(id, data, params)
remove(id, params)
```

Feathers supports [a common way](#) for querying, sorting, limiting and selecting find method calls as part of `params`, e.g. `{ query: { ... }, ... }`. Querying also applies to update, patch and remove method calls if the `_id` is set to `null`.

ProTip: The find method does not guarantee an order for the returned items.

Results

Run the program with `node ./examples/step/01/db-connector/1.js`. The console displays:

```
feathers-guide$ node ./examples/step/01/db-connector/1.js
created Jane Doe item
{ email: 'jane.doe@gmail.com',
  password: 'X2y6',
  role: 'admin',
  _id: '6Rq704RPYE02TdAn' }
created John Doe item
{ email: 'john.doe@gmail.com',
  password: 'i6He',
  role: 'user',
  _id: 'Q2bnbsBRf01ScqoqY' }
created Judy Doe item
{ email: 'judy.doe@gmail.com',
  password: '7jHw',
  role: 'user',
  _id: 'Tymf6Nailusd5MZD' }
find all items
[ { email: 'jane.doe@gmail.com',
  password: 'X2y6',
  role: 'admin',
  _id: '6Rq704RPYE02TdAn' },
  { email: 'john.doe@gmail.com',
    password: 'i6He',
    role: 'user',
    _id: 'Q2bnbsBRf01ScqoqY' },
  { email: 'judy.doe@gmail.com',
    password: '7jHw',
    role: 'user',
    _id: 'Tymf6Nailusd5MZD' } ]
```

Boilerplate. Feathers requires little boilerplate. It took only 15 lines of code to connect to a database.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

A REST API server

Our database connector will now function as a full fledged REST API server. We need only add a HTTP server to it.

HTTP servers. Feathers is currently tied into the popular HTTP server framework [Express](#). Future versions will support multiple frameworks, starting with [koa](#).

Working example

- Source code: [examples/step/01/rest/1.js](#) and [common](#)
- Run it: `node ./examples/step/01/rest/1.js`
- Compare with last page's [examples/step/01/db-connector/1.js](#): [Unified](#) | [Split](#)

Implementing a REST API server

This is our previous example with the database method calls removed, and with an Express server added.

```
// Example - Create REST API

const expressServerConfig = require('../common/expressServerConfig');
const expressMiddleware = require('../common/expressMiddleware');
const rest = require('feathers-rest');
const NeDB = require('nedb');
const path = require('path');
const service = require('feathers-nedb');

const app = expressServerConfig()
  .configure(rest())
  .configure(services)
  .configure(expressMiddleware);

const server = app.listen(3030);
server.on('listening', () => console.log(`Feathers application started on port 3030`));

function services() {
  this.use('/users', service({ Model: userModel() }));
}

function userModel() {
  return new NeDB({
    filename: path.join('examples', 'step', 'data', 'users.db'),
    autoload: true
  });
}
```

- See what changed: [Unified](#) | [Split](#)

The Express server `common/expressServerConfig.js` is configured as follows.

```
const bodyParser = require('body-parser');
const compress = require('compression');
const cors = require('cors');
const path = require('path');
const feathers = require('feathers');
```

```

module.exports = () => {
  const app = feathers()
    .use(compress())
    .options('*', cors())
    .use(cors())
    .use('/', feathers.static(path.join(__dirname, 'public')))
    .use(bodyParser.json())
    .use(bodyParser.urlencoded({ extended: true }));
  return app;
};

```

The Express middleware `common/expressMiddleware/index.js` handles logging, pages not found, and general errors.

```

'use strict';

const handler = require('feathers-errors/handler');
const notFound = require('../not-found-handler');
const logger = require('../logger');

module.exports = function() {
  const app = this;

  app.use(notFound());
  app.use(logger(app));
  app.use(handler());
};

```

Boilerplate. The server configuration and middleware are standard Express. They have little to do with Feathers other than to feed REST requests to it.

Running the server

We can now make REST API calls to the server.

In the previous example we created 3 user items and then printed the user file. We can now do the same thing, but using REST, with `curl` commands:

```

printf "\nPOST Jane Doe\n"
curl -H "Content-Type: application/json" -X POST -d '{"email":"jane.doe@gmail.com","password":"X2y6","role":"admin"}' http://localhost:3030/users
printf "\nPOST John Doe\n"
curl -H "Content-Type: application/json" -X POST -d '{"email":"john.doe@gmail.com","password":"i6He","role":"user"}' http://localhost:3030/users
printf "\nPOST Judy Doe\n"
curl -H "Content-Type: application/json" -X POST -d '{"email":"judy.doe@gmail.com","password":"7jHw","role":"user"}' http://localhost:3030/users
printf "\nGET all users\n"
curl -X GET http://localhost:3030/users

```

First, start the server by running `node ./examples/step/01/rest/1.js` on one terminal.

Then run the curl commands with `./examples/step/01/rest/curl-requests.sh` on another terminal.

Results

That console displays:

```
feathers-guide$ ./examples/step/01/rest/curl-requests.sh
```

```
POST Jane Doe
{"email":"jane.doe@gmail.com","password":"X2y6","role":"admin","_id":"sbkXv7LVkMhx1NyY"}
POST John Doe
{"email":"john.doe@gmail.com","password":"i6He","role":"user","_id":"uKhqOp4R4hABw9o0"}
POST Judy Doe
{"email":"judy.doe@gmail.com","password":"7jHw","role":"user","_id":"pvcmh9X2i9VZgqWJ"}
GET all users
[
  {"email":"judy.doe@gmail.com","password":"7jHw","role":"user","_id":"pvcmh9X2i9VZgqWJ"},
  {"email":"jane.doe@gmail.com","password":"X2y6","role":"admin","_id":"sbkXv7LVkMhx1NyY"},
  {"email":"john.doe@gmail.com","password":"i6He","role":"user","_id":"uKhqOp4R4hABw9o0"}
]
```

Feathers. REST API calls are automatically converted into Feathers database method calls like the `users.create()` and `users.find()` methods we used in the previous example. How's that for convenience?

Is anything wrong, unclear, missing?

[Leave a comment.](#)

A Feathers REST Client

We already have a Feathers REST API server from the previous example. Let's write a JavaScript frontend for it.

Working example

- Server code: [examples/step/01/rest/2.js](#)
- Client code: [common/public/rest.html](#) and [feathers-app.js](#)
- Start the server: `node ./examples/step/01/rest/2`
- Point the browser at: `localhost:3030/rest.html`
- Compare with last page's server [examples/step/01/rest/1.js](#): [Unified](#) | [Split](#)

Writing a server for Feathers client REST calls

[rest/2.js](#) , our server for Feathers REST clients, is exactly the same as [rest/1.js](#) , our previous server for HTTP REST API calls. **No new server code is required to handle Feathers REST clients.**

Compare the two: [Unified](#) | [Split](#).

Writing the frontend HTML

We'll soon see most of the frontend doesn't care if we're communicating with the server using REST or WebSockets. To keep things DRY, we are isolating code unique to REST in [common/public/rest.html](#).

```
<html>
<head>
  <title>Feathers REST client</title>
  <style>
    body {
      font-family: 'Helvetica Neue', 'Helvetica', 'Arial', 'sans-serif';
      font-weight: 400;
      font-size: 16px;
      color: #333;
    }
  </style>
</head>
<body>
  <h1>Feathers guide</h1>
  <h2>Feathers REST client</h2>
  <br />
  Open console to see results of <strong>feathers-rest</strong> calls.
  <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
  <script src="https://unpkg.com/feathers-client@^2.0.0/dist/feathers.js"></script>
  <script src="/serverUrl.js"></script>
  <script>
    const feathersClient = feathers()
      .configure(feathers.rest(serverUrl).fetch(fetch))
  </script>
  <script src="/feathers-app.js"></script>
</body>
</html>
```

- `https://cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js` loads a polyfill for `fetch` if required.
- `src="https://unpkg.com/feathers-client@^2.0.0/dist/feathers.js"` loads the Feathers client code.

- `src="/serverUrl.js"` loads the URL of the server. The default is `var serverUrl = 'http://localhost:3030';`. Change the value if you need to.
- `const feathersClient = feathers()` instantiates a Feathers client.
- `.configure(feathers.rest(serverUrl).fetch(fetch))` configures the client to use REST when communicating with the server. It points to the server, and passes the `fetch` instruction as the interface for fetching resources.
- `src="/feathers-app.js"` loads the main application.

Writing the Feathers frontend

Writing the HTML was actually the hard part. The frontend [common/public/feathers-app.js](#) is essentially the same as the server code we used in [Writing a Database Connector!](#)

```
const users = feathersClient.service('/users');

Promise.all([
  users.create({ email: 'jane.doe@gmail.com', password: '11111111', role: 'admin' }),
  users.create({ email: 'john.doe@gmail.com', password: '22222222', role: 'user' }),
  users.create({ email: 'judy.doe@gmail.com', password: '33333333', role: 'user' })
])
.then(results => {
  console.log('created Jane Doe item\n', results[0]);
  console.log('created John Doe item\n', results[1]);
  console.log('created Judy Doe item\n', results[2]);

  return users.find()
    .then(results => console.log('find all items\n', results));
})
.catch(err => console.log('Error occurred:', err));
```

- See what changed: [Unified | Split](#).

Feathers "a-ha!" moment. We can run **exactly** the same code on the frontend as on the server. We can code the frontend as if the database was sitting on it. That's part of the magic of Feathers, and it makes frontend development significantly simpler.

Results

The results in the console window of the browser are the same as they were running [Writing a Database Connector](#).

```
created Jane Doe item
Object {email: "jane.doe@gmail.com", password: "11111", role: "admin", _id: "8zQ7mXay3XqiqP35"}
created John Doe item
Object {email: "john.doe@gmail.com", password: "22222", role: "user", _id: "l9d0Txh0xk1h94gh"}
created Judy Doe item
Object {email: "judy.doe@gmail.com", password: "33333", role: "user", _id: "3BeFPGkduhM6mlwM"}
find all items
[Object, Object, Object]
  0: Object
    _id: "3BeFPGkduhM6mlwM"
    email: "judy.doe@gmail.com"
    password: "33333"
    role: "user"
  1: Object
    _id: "8zQ7mXay3XqiqP35"
    email: "jane.doe@gmail.com"
    password: "11111"
    role: "admin"
  2: Object
```

```
_id: "19d0Txh0xk1h94gh"  
email: "john.doe@gmail.com"  
password: "22222"  
role: "user"  
length: 3
```

Is anything wrong, unclear, missing?

[Leave a comment.](#)

A Feathers WebSocket Client

We already have a Feathers REST frontend. Its simple to convert that to one using WebSockets.

WebSockets. Feathers can use eight of the most popular WebSocket libraries. We will use the popular Socket.io in this guide.

Working example

- Server code: [examples/step/01/websocket/1.js](#)
- Client code: [common/public/socketio.html](#) and [feathers-app.js](#)
- Start the server: `node ./examples/step/01/websocket/1`
- Point the browser at: `localhost:3030/socketio.html`
- Compare with last page's server [examples/step/01/rest/2.js](#): [Unified](#) | [Split](#)
- Compare with last page's HTML [common/public/socketio.html](#) [Unified](#) | [Split](#)

Change the server to support clients using either Feathers REST or WebSocket calls

Add 2 lines to the server code so it supports either REST or WebSocket calls from the Feathers client.

```
const rest = require('feathers-rest');
const socketio = require('feathers-socketio'); // new

const app = expressServerConfig()
  .configure(rest())
  .configure(socketio()) // new
  .configure(services)
  .configure(middleware);
```

- See what changed: [Unified](#) | [Split](#)

Changing the HTML for Feathers client WebSocket calls

We replace the REST code we had in the HTML with the equivalent WebSocket code.

```
<script type="text/javascript" src="http://cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
<script src="https://unpkg.com/feathers-client@^2.0.0/dist/feathers.js"></script>
<script src="/socket.io.min.js"></script>
<script src="/serverUrl.js"></script>
<script>
  const socket = io(serverUrl);
  const feathersClient = feathers()
    .configure(feathers.socketio(socket))
</script>
<script src="/feathers-app.js"></script>
```

- See what changed: [Unified](#) | [Split](#)
- `src="/socket.io.min.js"` load the Socket.io client code.
- `const socket = io(serverUrl);` create a WebSocket.
- `.configure(feathers.socketio(socket))` configure Feathers client to use the WebSocket.

Changing the frontend code

We've already said that most of the Feathers frontend doesn't care if it's communicating with the server using REST or WebSockets. **No more changes are necessary.**

REST vs WebSockets. There is a huge technical difference involved in communicating via REST or WebSockets. Feathers hides this so you can get on with what's important rather than handling such details.

Results

And that's all there is to it. The results are identical to that for [A Feathers REST Client](#)

Is anything wrong, unclear, missing?

[Leave a comment.](#)

The Feathers "a-ha!" moment

Feathers is transport agnostic

A Feathers server automatically handles requests from

- HTTP REST clients,
- Feathers REST clients, or
- Feathers WebSocket clients.

Feathers resource management is platform agnostic

Identical database calls may be made on the server and on the frontend without you needing to do anything.

- This makes frontend development **significantly** easier.
- It allows you to share code between the server and the frontend.

The "a-ha!" moment

The "a-ha!" moment comes when you start to realize the significance of these features. Enormous amounts of boilerplate simply disappear.

You might now start to appreciate why people are enthusiastic about Feathers.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Hooks middleware

Linux pipes

One of the more powerful features of Linux is pipes, which has shaped its toolbox philosophy. A pipeline is a sequence of processes chained together, so that the output of each process feeds directly as input to the next one.

For example, to list files in the current directory (`ls`), retain only the lines of `ls` output containing the string "key" (`grep`), and view the result in a scrolling page (`less`), a user types the following into the command line of a terminal:

```
ls -1 | grep key | less
```

Express middleware

At the heart of pipes lies a design pattern: [Chain of Responsibility \(CoR\)](#). The CoR pattern uses a chain of objects to handle a request. Objects in the chain forward the request along the chain. Processing stops after an event is handled.



[Express middleware](#) uses the CoR pattern. You should be familiar with the following code if you've ever used Express. A HTTP request is handled by each of these Express functions in sequence.

```
app.use(cors());
app.use(helmet());
app.use(compress());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(favicon(path.join(app.get('public'), 'favicon.ico')));
app.use('/', feathers.static(app.get('public')));
```

The CoR pattern promotes the idea of [loose coupling](#) and it lets us combine simple functions to build solutions for specific needs.

Feathers hooks

Applications are about more than the reading and writing of raw database items. Application-specific logic often needs to run before and after service calls.

You will implement most of your business requirements with [service hooks](#), which are middleware functions that run for each service method. Feathers calls the data passed between these hooks the `context` object (in order not to confuse them with the HTTP `request` object).

Hook middleware is organized like this:



and the corresponding Feathers code would be:

```
const messagesHooks = {
  before: {
    create: [ hook31(), hook32() ]
  },
  after: {
    create: hook35()
  }
};
const messages = app.service('messages');
messages.hooks(messagesHooks);
```

You can see that a series of hooks are run between the service call and the actual call to the database. These hooks may, for example:

- Ensure the user is authenticated,
- Ensure the user is allowed to perform this operation,
- Validate the data for the service call. i.e. the data in the `context` object,
- Update the record's `updatedAt` value, thus modifying the `context` object,
- Perhaps not allow the service call to proceed, or return a specified response for it.

ProTip Hooks may be synchronous or async (using promises or `async/await`). The next hook will run only when the current one finishes (sync) or resolves (async), so hooks are always run sequentially.

A series of hooks is also run after the actual call to the database, if that call was successful. These hooks may:

- Populate the response with related information, e.g. information about the user who created the returned record,
- Remove information for security reasons, e.g. the user password.

You can implement all your business logic related to service calls with hooks.

ProTip The DB call is middleware too! It uses the information in the `context` object to call the database, and then updates `context` with the result.

ProTip Services are usually database adapters, but they need not be. You can create a service which writes to the server log for example. A client could post logs using this service. Hooks may be defined for all services, regardless of their type.

ProTip Some people may call the `context` object the `hook` object. The two terms are interchangeable.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

All About Hooks

Method hooks

Each service method may have its own hooks. After all, you'll likely need to do something different for `create` than for a `find`.



The Feathers code would be:

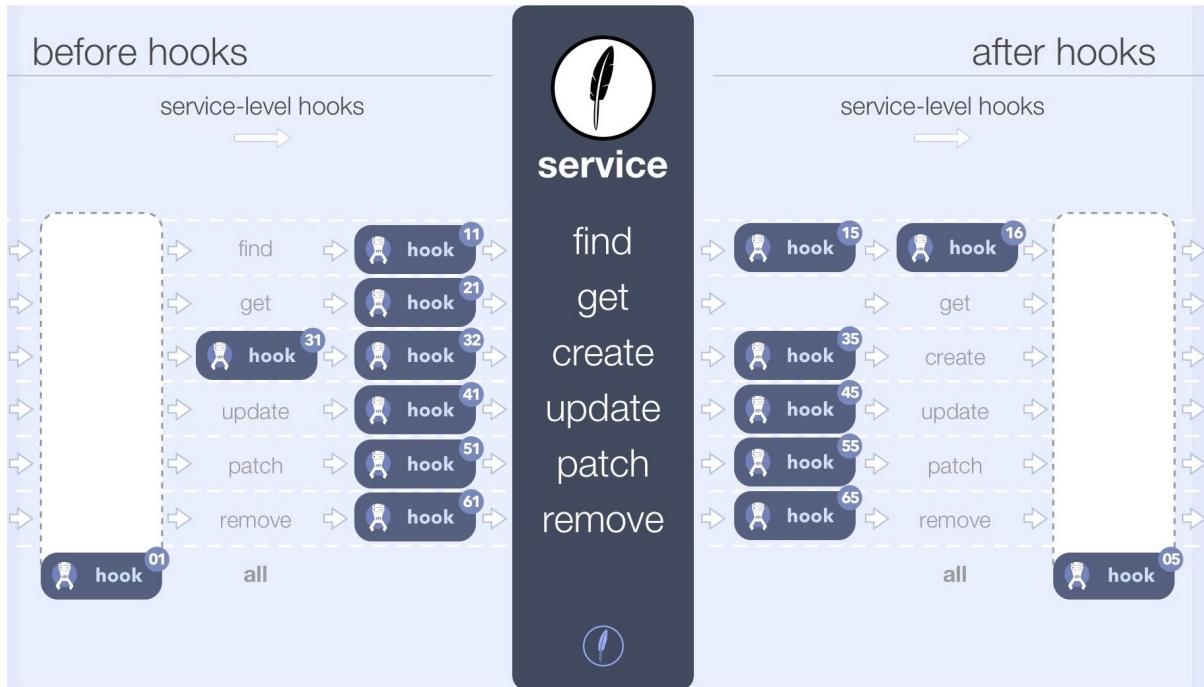
```
const messagesHooks = {
  before: {
    find: hook11(),
    get: hook21(),
    create: [ hook31(), hook32() ],
    update: hook41(),
    patch: hook51(),
    remove: hook61(),
  },
  after: {
    find: [ hook15(), hook16() ],
    create: hook35(),
    update: hook45(),
    patch: hook55(),
    remove: hook65(),
  }
};
const messages = app.service('messages');
messages.hooks(messagesHooks);
```

ProTip: The Feathers service call handler expects the functions it calls to have the signature `context => {}`.

So if you have such a hook you would code `{ before: { all: myHook } }`. You however commonly want to pass params to the hook, such as which field in the record to delete. So you need to use a signature like `params => context => { /* use params */ }` and code `all: myHook(params)`. The common hooks and the Feathers community tend to use the latter signature for consistency. So it would be best to set up your hooks to use `myHook()`.

Service hooks

Some hooks, such as authentication, may need to be run for every method. You can specify them once rather than repeating them for every method.

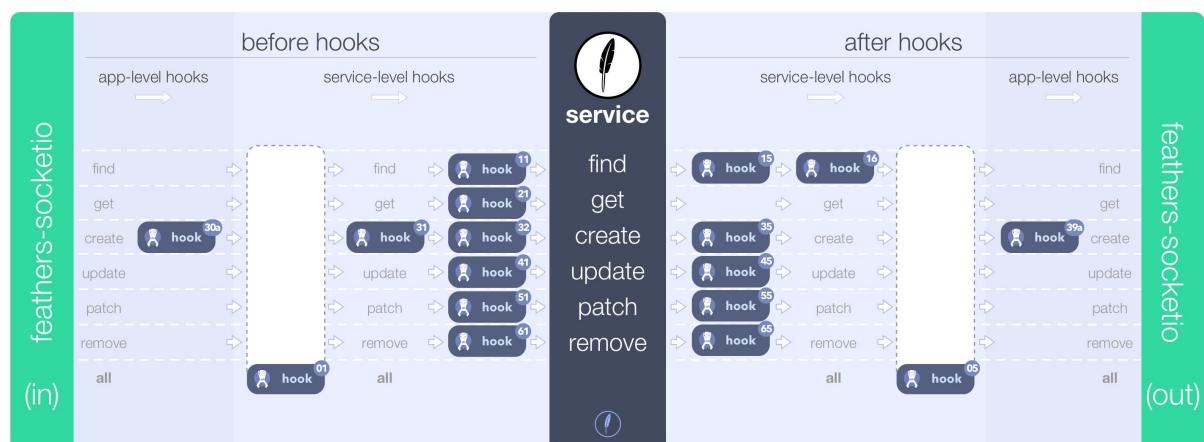


Your Feathers code would *additionally* include:

```
const messagesHooks = {
  before: {
    all: hook01(),
  },
  after: {
    all: hook05(),
  }
};
```

App hooks

You may want to run some hooks for every service. The [Feathers profiler](#), for example, adds before and after hooks to time each service call.



The Feathers code for these application level hooks would be:

```
app.hooks({
  before: {
    create: hook30a()
```

```

},
after: {
  create: hook39a()
},
});

```

Errors and error hooks

Errors may be thrown inside hooks - by JavaScript, by the Feathers database adapter, or by your own code.

Your hook can, for example, return a formatted message as follows:

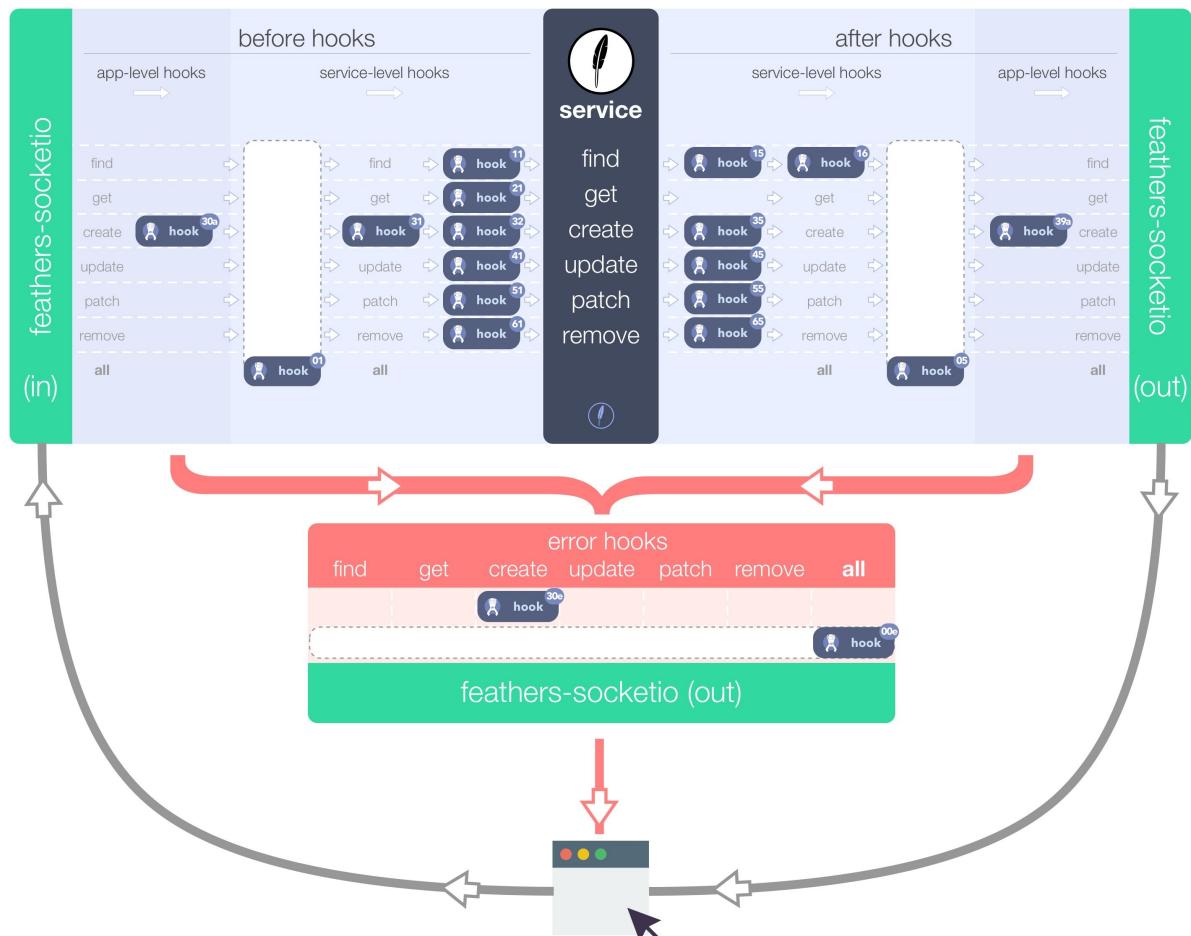
```

// On server
const errors = require('feathers-errors');
throw new errors.BadRequest('Invalid request', {errors: {email: 'Invalid Email'}}); // inside hook

// On client
messages.create(...)
  .then(data => ...)
  .catch(err => {
    console.log(err.messages); // Invalid request
    console.log(err.errors.email); // Invalid Email
  });

```

You can optionally deal with thrown errors in the service:



Your Feathers code would *additionally* include:

```
app.hooks({
  error: {
    all: hook00e(),
    create: hook30e()
  }
});
```

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Hooks, part 1

Common hooks

Hooks allows us to combine simple functions to build complicated solutions. Most hooks will be general in nature and they may be used with multiple services.

Feathers comes with a set of [commonly useful hooks](#). Let's work with some of them.

Working example

- Server code: [examples/step/01/hooks/1.js](#)
- Client code: [common/public/rest.html](#) and [feathers-app.js](#)
- Start the server: `node ./examples/step/01/hooks/1`
- Point the browser at: `localhost:3030/rest.html`
- Compare with last page's server [examples/step/01/hooks/1.js](#): [Unified](#) | [Split](#)

Writing hooks

Let's add some hooks to the server we've used with the Feathers REST and WebSocket clients.

```
const authHooks = require('feathers-authentication-local').hooks;
const hooks = require('feathers-hooks');
const commonHooks = require('feathers-hooks-common');

const app = httpServerConfig()
  .configure(hooks())

function services() {
  this.configure(user);
}

function user() {
  const app = this;

  app.use('/users', service({ Model: userModel() }));
  const userService = app.service('users');

  const { validateSchema, setCreatedAt, setUpdatedAt, unless, remove } = commonHooks;

  userService.before({
    create: [
      validateSchema(userSchema(), Ajv), authHooks.hashPassword(), setCreatedAt(), setUpdatedAt()
    ]
  });
  userService.after({
    all: unless(hook => hook.method === 'find', remove('password')),
  });
}

function userSchema() {
  return {
    title: 'User Schema',
    $schema: 'http://json-schema.org/draft-04/schema#',
    type: 'object',
    required: [ 'email', 'password', 'role' ],
    additionalProperties: false,
  }
}
```

```

    properties: {
      email: { type: 'string', maxLength: 100, minLength: 6 },
      password: { type: 'string', maxLength: 30, minLength: 8 },
      role: { type: 'string' }
    }
  };
}

```

- See what changed: [Unified | Split](#)

- **.configure(hooks())**

We include support for hooks in the configuration.

- **this.configure(user);**

The user service is now more complex, so we configure it on its own.

- **const { validateSchema, setCreatedAt, setUpdatedAt, unless, remove } = commonHooks;**

Feathers comes with a library of useful hooks. Here we get some common hooks from `feathers-hooks-common`. More specialized hooks come bundled with their specialized packages.

- **userService.before({ ... });**

These hooks will be run before the operation on the database.

- **create: [...]**

These hooks will be run before all `create` operations on the database. `all` (all service methods), `get`, `update`, `patch`, `remove`, `find` may also be included.

- **validateSchema(userSchema(), Ajv)**

Validate the data we are to add using `Ajv`. The service's `JSON schema` is provided by `function userSchema`.

There are [good tutorials](#) on validating data with JSON schema.

- **authHooks.hashPassword()**

The data has a `password` field. This specialized authentication hook will replace it with a hashed version so the password may be stored safely.

bcrypt. Feathers hashes passwords using `bcrypt`. bcrypt has the best kind of repute that can be achieved for a cryptographic algorithm: it has been around for quite some time, used quite widely, "attracted attention", and yet remains unbroken to date. ([Reference.](#))

JSON webtoken. Feathers Authentication uses JSON webtoken (JWT) for secure authentication between a client and server as opposed to cookies and sessions. The cookies vs token debate [favors token-based authentication](#). The avoidance of sessions makes Feathers apps more easily scalable.

- **setCreatedAt(), setUpdatedAt()**

These hooks add `createdAt` and `updatedAt` properties to the data.

- **userService.after({ ... });**

These hooks are run after the operation on the database. They act on all the results returned by the operation.

- **unless(hook => hook.method === 'find', remove('password'))**

- `hook => hook.method === 'find'` returns true if the database operation was a `find`. All hooks are passed a `hook object` which contains information about the operation.
- `remove('password')` removes the `password` property from the results.
- `unless(predicate, ...hooks)` runs the hooks if the predicate is false.

It's not secure to return the encoded password to the client, so this hook removes it. We have made an exception for the `find` operation because we want you to see something in the results.

Hooks. We are now doing some processing typical of apps. Before we add a new user, we verify the data, encode the password, and add `createdAt` plus `updatedAt` properties. We remove the `password` field before we return the results to the client.

Hooks

Many of your common needs are already handled by hooks in the [common hooks library](#). This may significantly reduce the code you need to write.

Hooks are just small middleware functions that get applied before and after a service method executes.

Hooks are transport independent. It does not matter if the service request come through HTTP REST, Feathers REST, Feathers WebSockets, or any other transport Feathers may support in the future.

Most hooks can be used with any service. This allows you to easily decouple the actual service logic from things like authorization, data pre-processing (sanitizing and validating), data post processing (serialization), or sending notifications like emails or text messages after something happened.

You can swap databases with minimal application code changes. You can also share validations for multiple databases in the same app, across multiple apps, and with your client.

Results

The browser console displays

```
created Jane Doe item
Object {email: "jane.doe@gmail.com", role: "admin", createdAt: "2017-05-31T08:33:07.642Z", updatedAt: "2017-05-31T08:33:07.643Z", _id: "VNSm7SxZnMeVxN6Z"}
created John Doe item
Object {email: "john.doe@gmail.com", role: "user", createdAt: "2017-05-31T08:33:07.643Z", updatedAt: "2017-05-31T08:33:07.643Z", _id: "SHyBbGehbEiZGpQS"}
created Judy Doe item
Object {email: "judy.doe@gmail.com", role: "user", createdAt: "2017-05-31T08:33:07.645Z", updatedAt: "2017-05-31T08:33:07.645Z", _id: "2zFj0CoGjczuQP5B"}
find all items
[Object, Object, Object]
0: Object
  email: "judy.doe@gmail.com"
  password: "$2a$10$TnuSw.09Jfss61BUFB0TTeT9dD0dtSX00C.19vX484eICygo7xXMe"
  role: "user"
  createdAt: "2017-05-31T08:33:07.645Z"
  updatedAt: "2017-05-31T08:33:07.645Z"
```

```
_id: "2zFj0CoGjczuQP5B"
1: Object
  email: "john.doe@gmail.com"
  password: "$2a$10$jI7lypIIiImLw7Kf9mN.N0faRkdP0sM0CeR1anH0J/f6p3fI9s2nu"
  role: "user"
  createdAt: "2017-05-31T08:33:07.643Z"
  updatedAt: "2017-05-31T08:33:07.643Z"
  _id: "SHyBbGehbEiZGpQS"
2: Object
  email: "jane.doe@gmail.com"
  password: "$2a$10$Dx3e/3vSn4Eq2MRvKAUGYeUMWMUeuTG6PCJpxx9/Uyov5IZRb1B.6"
  role: "admin"
  createdAt: "2017-05-31T08:33:07.642Z"
  updatedAt: "2017-05-31T08:33:07.643Z"
  _id: "VNSm7SxZnMeVxN6Z"
length: 3
```

- `createdAt` and `updatedAt` have been added to the items.
- `password` is not included in the data returned for the create and delete operations.
- An encoded `password` is included for the find operation, because of the special coding we included in the example.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Hooks, part 2

If you have an archive of stock movements, you cannot simply delete a stock item you no longer want to keep. You would not be able to properly present historical data if you did so.

The solution is to keep the data but mark it as deleted. We can ignore the `deleted` flag when we know we are accessing historical, and possibly deleted, items. Otherwise we want the application to act as if the item didn't exist.

It would be fairly complex to implement **soft delete** support yourself, however its easy to do using the `softDelete` hook.

Working example

- Server code: [examples/step/01/hooks/2.js](#)
- Client code: [common/public/rest-del.html](#) and [common/public/feathers-app-del.js](#)
- Start the server: `node ./examples/step/01/hooks/2`
- Point the browser at: `localhost:3030/rest-del.html`
- Compare with last page's server [hooks/1.js.](#): [Unified](#) | [Split](#)

Using softDelete

We need to make just one change to our previous server example. We use the `when` hook to run the `softDelete` hook if the service method is not find.

```
const {
  softDelete, when, // new
  setCreatedAt, setUpdatedAt, unless, remove
} = commonHooks;
// ...
userService.before({
  all: when(hook => hook.method !== 'find', softDelete()), // new
  create: [ /* ... */ ]
});
```

- See what changed: [Unified](#) | [Split](#)

The results

The browser console displays

```
created Jane Doe item
Object {email: "jane.doe@gmail.com", role: "admin", createdAt: "2017-05-31T08:41:48.640Z", updatedAt: "2017-05-31T08:41:48.640Z", _id: "CfLheOpJ3rve1IPh"}
created John Doe item
Object {email: "john.doe@gmail.com", role: "user", createdAt: "2017-05-31T08:41:48.623Z", updatedAt: "2017-05-31T08:41:48.623Z", _id: "sQy34FUrDC8gJOUR"}
created Judy Doe item
Object {email: "judy.doe@gmail.com", role: "user", createdAt: "2017-05-31T08:41:48.641Z", updatedAt: "2017-05-31T08:41:48.641Z", _id: "eKNolHDB06qXH2MU"}
created Jack Doe item
Object {email: "jack.doe@gmail.com", role: "user", createdAt: "2017-05-31T08:41:48.641Z", updatedAt: "2017-05-31T08:41:48.641Z", _id: "5iQCl2oDLbVXMfHo"}
deleted Jack Doe item
```

```

Object
  email: "jack.doe@gmail.com",
  role: "user",
  deleted: true
  createdAt: "2017-05-31T08:41:48.641Z",
  updatedAt: "2017-05-31T08:41:48.641Z",
  _id: "5iQC12oDLbVXMfHo"
find all items
[Object, Object, Object, Object]
  0: Object
    email: "jack.doe@gmail.com"
    role: "user"
    password: "$2a$10$So9MhiVGW.P31CZnUefXX0cuacwKMm7nTgCAPSBZB9r010how.X.G"
    deleted: true
    createdAt: "2017-05-31T08:41:48.641Z"
    updatedAt: "2017-05-31T08:41:48.641Z"
    _id: "5iQC12oDLbVXMfHo"
  1: Object
    email: "jane.doe@gmail.com"
    password: "$2a$10$TAz6SD6WxEostxvCNMOubuEY68pS8Jv9pLvrrgCiWTI0jIs3yIl0."
    role: "admin"
    createdAt: "2017-05-31T08:41:48.640Z"
    updatedAt: "2017-05-31T08:41:48.640Z"
    _id: "CfLheOpJ3rve1IPh"
  2: Object
    email: "judy.doe@gmail.com"
    password: "$2a$10$GvUEJfpTQLGY8JKTuH8yeeML9auvLo1IGDVyGF00ImZ0Nuxtd7uji"
    role: "user"
    createdAt: "2017-05-31T08:41:48.641Z"
    updatedAt: "2017-05-31T08:41:48.641Z"
    _id: "eKNolHDB06qXH2MU"
  3: Object
    email: "john.doe@gmail.com"
    password: "$2a$10$MX0LJerCfLoGx31mGh2x0eR7CyE2t2STeHhpcV9vYbpD3m8i80Z.S"
    role: "user"
    createdAt: "2017-05-31T08:41:48.623Z"
    updatedAt: "2017-05-31T08:41:48.623Z"
    _id: "sQy34FUUrDC8gJOUR"
length: 4

```

- The result returned when the Jack Doe item was deleted contains `deleted: true`.
- The results returned for find also contain `deleted: true` for Jack Doe because of how we conditioned the `softDelete` hook.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Writing Your Own Hooks

Hook function template

Hook functions should be written like this:

```
// Outer function initializes the hook function
function myHook(options) {
  // The hook function itself is returned.
  return context => {
    // You can use the options param to condition behavior within the hook.
  };
}
```

Feathers calls the inner function with the [context object](#).

`context.result` is an object or array for all method calls other than `find`. It is [an object](#) if the `find` is paginated. Otherwise it is an array.

The hook function may either return synchronously or it may return a Promise. The return value (`sync`) or resolved value from the Promise (`async`) may be either a new context object, or `undefined`.

ProTip The context object is not changed if `undefined` is returned.

ProTip Mutating the `context` param inside a hook function without returning it does not change the context object passed to the next hook.

Let's review the source of some of the [common hooks](#) to learn how to write our own.

debug source

`debug` logs the context to the console.

```
export default function (msg) {
  return context => {
    console.log(`* ${msg} || ''\n  type:${context.type}, method: ${context.method}`);
    if (context.data) { console.log('data:', context.data); }
    if (context.params && context.params.query) { console.log('query:', context.params.query); }
    if (context.result) { console.log('result:', context.result); }
    if (context.error) { console.log('error', context.error); }
  };
}
```

This hook is straightforward, simply displaying some of the `context` object properties. The context object does not change as the inner hook function returns `undefined` by default.

`debug` is great for debugging other hooks. Once you place this hook before and after the hook under test, you'll see the context object the test hook received, and what it returned.

This example

- Shows several context properties.
- Leaves the context object unchanged with a sync `return`.

disableMultiItemChange source

`disableMultiItemChange` disables update, patch and remove methods from using `null` as an id, e.g. `remove(null)`. A `null` id affects all the items in the DB, so accidentally using it may have undesirable results.

```
import errors from 'feathers-errors';
import checkContext from './check-context';

export default function () {
  return function (context) {
    checkContext(context, 'before', ['update', 'patch', 'remove'], 'disableMultiItemChange');

    if (context.id === null) {
      throw new errors.BadRequest(
        `Multi-record changes not allowed for ${context.path} ${context.method}. (disableMultiItemChange)`
      );
    }
  };
}
```

Some hooks may only be used `before` or `after`; some may be used only with certain methods. The `checkContext` utility checks the hook function is being used properly.

This hook throws an error that will be properly returned to the client.

```
service.patch(null, data, { query: { dept: 'acct' } })
  .then(data => ...)
  .catch(err => {
    console.log(err.message); // Multi-record changes not allowed for ...
  });
});
```

This example

- Introduces `checkContext`.
- Shows how to throw an error in hooks.

pluckQuery source

`pluckQuery` discards all fields from the query params except for the specified ones. This helps sanitize the query.

```
import _pluck from '../common/_pluck';
import checkContext from './check-context';

export default function (...fieldNames) {
  return context => {
    checkContext(context, 'before', null, 'pluckQuery');

    const query = (context.params || {}).query || {};
    context.params.query = _pluck(query, fieldNames);

    return context;
  };
}
```

The `_pluck` utility, given an object and an array of property name, returns an object consisting of just those properties. The property names may be in dot notation, e.g. `destination.address.city`.

The context object is modified and returned, thus modifying what context is passed to the next hook.

This example

- Modifies and synchronously returns the context object.
- Introduces `_pluck`.

pluck source

`pluck` discards all fields except for the specified ones, either from the data submitted or from the result. If the data is an array or a paginated find result the hook will remove the field(s) for every item.

```
import _pluck from '../common/_pluck';
import checkContextIf from './check-context-if';
import getItems from './get-items';
import replaceItems from './replace-items';

export default function (...fieldNames) {
  return context => {
    checkContextIf(context, 'before', ['create', 'update', 'patch'], 'pluck');

    if (context.params.provider) {
      replaceItems(context, _pluck(getItems(context), fieldNames));
    }

    return context;
  };
}
```

The `getItems` utility returns the items in either `hook.data` or `hook.result` depending on whether the hook is being used as a before or after hook. `hook.result.data` or `hook.result` is returned for a `find` method.

The returned items are always an array to simplify further processing.

The `replaceItems` utility is the reverse of `getItems`, returning the items where they came from.

This example

- Introduces the convenient `getItems` and `replaceItems` utilities.

Throwing an error - disableMultiItemChange source

You will, sooner or later, want to return an error to the caller, skipping the DB call. You can do this by throwing a `Feathers` error.

`disableMultiItemChange` disables update, patch and remove methods from using null as an id.

```
import errors from 'feathers-errors';
import checkContext from './check-context';

export default function () {
  return function (context) {
    checkContext(context, 'before', ['update', 'patch', 'remove'], 'disableMultiItemChange');

    if (context.id === null) {
      throw new errors.BadRequest(
        `Multi-record changes not allowed for ${context.path} ${context.method}. (disableMultiItemChange)`
      );
    }
  };
}
```

Feathers errors are flexible, containing [useful fields](#). Of particular note are:

- `className` returns the type of error, e.g. `not-found`. Your code can check this field rather than the text of the error message.
- `errors` can return error messages for individual fields. You can customize the format to that expected by your client-side forms handler.

```
throw new errors.BadRequest('Bad request.', { errors: {
  username: 'Already in use', password: 'Must be at least 8 characters long'
}});
```

This example

- Shows how to stop a method call by throwing an error.

Returning a result

Assume that for a service with static data the record is added to `cache` whenever a `get` call has completed. We can then potentially improve performance for future `get` calls by checking if we already have the record.

```
import { checkContext } from 'feathers-hooks-common';

export default function (cache) {
  return context => {
    checkContext(context, 'before', ['get'], 'memoize');

    if (context.id in cache) {
      context.result = cache[context.id];
      return context;
    }
  };
}
```

Feathers will not make the database call if `hook.result` is set. Any remaining before and after hooks are still run.

Should this hook find a cached record, placing it in `hook.result` is the same as if the database had returned the record.

This example

- Shows how `before` hooks can determine the result for the call.

Simple async hook

Now that we've covered synchronous hooks, let's look at async ones.

Here is a simple before hook which calls an async function. That function is supposed to determine if the values in `context.data` are valid.

```
import errors from 'feathers-errors';

export default function (validator) {
  return context => {
    return validator(context.data)
      .then(() => context)
      .catch(errs => {
        throw new errors.BadRequest('Validation error', { errors: errs });
      });
  };
}
```

```
}
```

The hook either returns a Promise which resolves to the existing context object, or it throws with an error object contains the errors found.

The hook after this one will not run until this Promise resolves and the hook logically ends.

ProTip Perhaps the most common error made when writing async hooks is to *not return* the Promise. The hook will not work as expected with `validator(context.data)`. It must be `return validator(context.data)`.

This example

- Shows how to code async hooks.

Calling a service

Here is an after hook which attaches user info to one record (for simplicity).

```
export default function () {
  return context => {
    const service = context.app.service('users');
    const item = getItems(context)[0];

    if (item.userId !== null && item.userId !== undefined) {
      return service.get(item.userId, context.params)
        .then(data => {
          item.user = data;
          return context;
        })
        .catch(() => context);
    }
  };
}
```

- `context.app` is the Feathers app, so `context.app.service(path/to/service)` returns that service.
- The hook returns a Promise which resolves to a mutated context, or
- the hook returns synchronously without modifying the context if there is no `userId`.

It's important that `context.params` is used in the `get`. You always need to consider `params` when calling a service within a hook. If you don't assign a value, the `get` will run as being called on the server (it is being called by the server after all) even if the method call causing the hook to be run originated on a client.

This may not be OK. The user password may be returned when a user record is read by the server, but you would not want a client to have access to it.

This hook has taken a simple approach, passing along the `context.params` of the method call. Thus the `get` is run with the same `context.provider` (e.g. "socketio", "rest", `undefined`), `context.authenticated`, etc. as the method call.

This is often satisfactory and, if not, the next example contains something more comprehensive.

ProTip Always consider `params` when doing service calls within a hook.

An interesting detail is shown here: `replaceItems` is never called. The array returned by `getItems` contains the same objects as those in the context. So changing an object in the array changes that object in the context. This is similar to:

```
const foo = { name: 'John' };
const bar = [ foo ];
bar[0].project = 'Feathers';
```

```
console.log(foo); // { name: 'John', project: 'Feathers' }
```

This example

- Shows how to call a service.
- Shows how to deal with `params` in such calls.
- Talks about using `getItems` with mutations.

stashBefore source

`stashBefore` saves the current value of record before mutating it.

```
import errors from 'feathers-errors';
import checkContext from './check-context';

export default function (prop) {
  const beforeField = prop || 'before';

  return context => {
    checkContext(context, 'before', ['get', 'update', 'patch', 'remove'], 'stashBefore');

    if (context.id === null || context.id === undefined) {
      throw new errors.BadRequest('Id is required. (stashBefore)');
    }

    if (context.params.query && context.params.query.$disableStashBefore) {
      delete context.params.query.$disableStashBefore;
      return context;
    }

    const params = (context.method === 'get') ? context.params : {
      provider: context.params.provider,
      authenticated: context.params.authenticated,
      user: context.params.user
    };

    params.query = params.query || {};
    params.query.$disableStashBefore = true;

    return context.service.get(context.id, params)
      .then(data => {
        delete params.query.$disableStashBefore;

        context.params[beforeField] = data;
        return context;
      })
      .catch(() => context);
  };
}
```

It's more complicated to call the hook's current service than to call another service. Let's look at some of the code in this hook.

This is what the hook returns.

```
return context.service.get(context.id, params)
  .then(data => {
    delete params.query.$disableStashBefore;

    context.params[beforeField] = data;
    return context;
})
```

```
.catch(() => context);
```

- `context.service` is always the current service.
- `context.service.get()` is an async call, and it returns a Promise.
- The hook returns that Promise, so its an async hook. The next hook will only run once this Promise resolves.
- The data obtained by the `get` is placed into `context.params`.
- We can see the Promise will always resolve to `context`.

In summary, the hook will `get` the record being mutated by the call, will place that record in `context.params`, and will return the possibly modified `context`. The method call will continue as if nothing has happened.

`stashBefore` does not use `context.params` in the `get` call as `context.params` may be inappropriate if, for example, you are using Sequelize and the method call includes parameters that are passed through to Sequelize. What may be appropriate for an `update` may not be acceptable for a `get`.

```
const params = context.method === 'get' ? context.params : {
  provider: context.params.provider,
  authenticated: context.params.authenticated,
  user: context.params.user
};
```

- On a `get` call we will use the same `params` for our inner `get`.
- On other calls, we use something "safe".
 - We copy over `provider` so our inner `get` acts like it has the same transport.
 - We copy standard authentication values for auth hooks.

Will this satisfy every use case? No, but it will satisfy most. You can always fork the hook and customize it.

There is one more thing to consider. The `stashBefore` hook will run again when we call the inner `get`. This will cause a recursion of inner `get` calls unless we do something.

```
if (context.params.query && context.params.query.$disableStashBefore) {
  delete context.params.query.$disableStashBefore;
  return context;
}
// ...
params.query = params.query || {};
params.query.$disableStashBefore = true;
// ...
delete params.query.$disableStashBefore;
```

We set a flag to show we are calling the inner `get`. `stashBefore` will see the flag when it runs for that inner `get` and exit, preventing recursion.

ProTip Its not uncommon to indicate what state operations are in by setting flags in `params`.

This example

- Shows how to call the current service.
- Discusses how to handle `params` for service calls.
- Shows how to prevent recursion.

iff, when, else

Conditional hooks like `iff(predicate, hook1, hook2).else(hook3, hook4)` can be very useful.

It's easy to write your own predicates. They are functions with a signature of `context => boolean`, which receive the context as a parameter and return either a boolean (synchronous) or a Promise which resolves to a boolean.

You can combine predicates provided with the common hooks, such as `isProvider` ([source](#)). You can write your own, or mix and match.

```
iff (hook => !isProvider('service')(hook) && hook.params.user.security >= 3, ...)
```

The `isNot` conditional utility ([source](#)) is useful because it will negate either a boolean or a Promise resolving to a boolean.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Testing Hooks

Self contained hooks

Testing hooks that do not depend on services or other hooks is straight forward. Create a `context` object, call the inner hook function, and check the returned `context`.

Here is part of the mocha test for `disableMultiItemChange`.

```
import { assert } from 'chai';
import { disableMultiItemChange } from '../../src/services';

var hookBefore;

['update', 'patch', 'remove'].forEach(method => {
  describe(`services disableMultiItemChange - ${method}`, () => {
    beforeEach(() => {
      hookBefore = {
        type: 'before',
        method,
        params: { provider: 'rest' },
        data: { first: 'John', last: 'Doe' },
        id: null
      };
    });
    it('allows non null id', () => {
      hookBefore.id = 1;

      const result = disableMultiItemChange()(hookBefore);
      assert.equal(result, undefined);
    });

    it('throws on null id', () => {
      hookBefore.id = null;

      assert.throws(() => { disableMultiItemChange()(hookBefore); });
    });

    it('throws if after hook', () => {
      hookBefore.id = 1;
      hookBefore.type = 'after';

      assert.throws(() => { disableMultiItemChange()(hookBefore); });
    });
  });
});
```

Here is part of the mocha test for `pluck`.

```
import { assert } from 'chai';
import hooks from '../../src/services';

var hookBefore;
var hookAfter;
var hookFindPaginate;
var hookFind;

describe('services pluck', () => {
  describe('plucks fields', () => {
    beforeEach(() => {
```

```

hookBefore = {
  type: 'before',
  method: 'create',
  params: { provider: 'rest' },
  data: { first: 'John', last: 'Doe' } };
hookAfter = {
  type: 'after',
  method: 'create',
  params: { provider: 'rest' },
  result: { first: 'Jane', last: 'Doe' } };
hookFindPaginate = {
  type: 'after',
  method: 'find',
  params: { provider: 'rest' },
  result: {
    total: 2,
    data: [
      { first: 'John', last: 'Doe' },
      { first: 'Jane', last: 'Doe' }
    ]
  } };
hookFind = {
  type: 'after',
  method: 'find',
  params: { provider: 'rest' },
  result: [
    { first: 'John', last: 'Doe' },
    { first: 'Jane', last: 'Doe' }
  ]
};

it('updates hook before::create', () => {
  hooks.pluck('last')(hookBefore);
  assert.deepEqual(hookBefore.data, { last: 'Doe' });
});

it('updates hook after::find with pagination', () => {
  hooks.pluck('first')(hookFindPaginate);
  assert.deepEqual(hookFindPaginate.result.data, [
    { first: 'John' },
    { first: 'Jane' }
  ]);
});

it('updates hook after::find with no pagination', () => {
  hooks.pluck('first')(hookFind);
  assert.deepEqual(hookFind.result, [
    { first: 'John' },
    { first: 'Jane' }
  ]);
});

it('updates hook after', () => {
  hooks.pluck('first')(hookAfter);
  assert.deepEqual(hookAfter.result, { first: 'Jane' });
});

it('does not update when called internally on server', () => {
  hookAfter.params.provider = '';
  hooks.pluck('last')(hookAfter);
  assert.deepEqual(hookAfter.result, { first: 'Jane', last: 'Doe' });
});

it('does not throw if field is missing', () => {
  const hook = {
    type: 'before',
    method: 'create',
    params: { provider: 'rest' },

```

```

        data: { first: 'John', last: 'Doe' } };
    hooks.pluck('last', 'xx')(hook);
    assert.deepEqual(hook.data, { last: 'Doe' });
});

it('does not throw if field is undefined', () => {
  const hook = {
    type: 'before',
    method: 'create',
    params: { provider: 'rest' },
    data: { first: undefined, last: undefined } };
  hooks.pluck('first')(hook);
  assert.deepEqual(hook.data, {}); // todo note this
});

it('does not throw if field is null', () => {
  const hook = {
    type: 'before',
    method: 'create',
    params: { provider: 'rest' },
    data: { first: null, last: null } };
  hooks.pluck('last')(hook);
  assert.deepEqual(hook.data, { last: null });
});
});
});

```

Hooks requiring a Feathers app

Some hooks call services, or they depend on other hooks running. Its much simpler to create a Feathers app plus a memory-backed service than to try to mock them out.

Here is part of the mocha test for `stashBefore`.

```

const assert = require('chai').assert;
const feathers = require('feathers');
const memory = require('feathers-memory');
const feathersHooks = require('feathers-hooks');
const { stashBefore } = require('../src/services');

const startId = 6;
const storeInit = {
  '0': { name: 'Jane Doe', key: 'a', id: 0 },
  '1': { name: 'Jack Doe', key: 'a', id: 1 },
  '2': { name: 'John Doe', key: 'a', id: 2 },
  '3': { name: 'Rick Doe', key: 'b', id: 3 },
  '4': { name: 'Dick Doe', key: 'b', id: 4 },
  '5': { name: 'Dork Doe', key: 'b', id: 5 }
};
let store;
let finalParams;

function services () {
  const app = this;
  app.configure(users);
}

function users () {
  const app = this;
  store = clone(storeInit);

  app.use('users', memory({
    store,
    startId
  }));
}

```

```

app.service('users').before({
  all: [
    stashBefore(),
    context => {
      finalParams = context.params;
    }
  ]
});
}

describe('services stash-before', () => {
  let app;
  let users;

  beforeEach(() => {
    finalParams = null;

    app = feathers()
      .configure(feathersHooks())
      .configure(services);

    users = app.service('users');
  });

  ['get', 'update', 'patch', 'remove'].forEach(method => {
    it(`stash on ${method}`, () => {
      return users[method](0, {})
        .then(() => {
          assert.deepEqual(finalParams.before, storeInit[0]);
        });
    });
  });

  ['create', 'find'].forEach(method => {
    it(`throws on ${method}`, done => {
      users[method]({})
        .then(() => {
          assert(false, 'unexpectedly successful');
          done();
        })
        .catch(() => {
          done();
        });
    });
  });
}

function clone (obj) {
  return JSON.parse(JSON.stringify(obj));
}

```

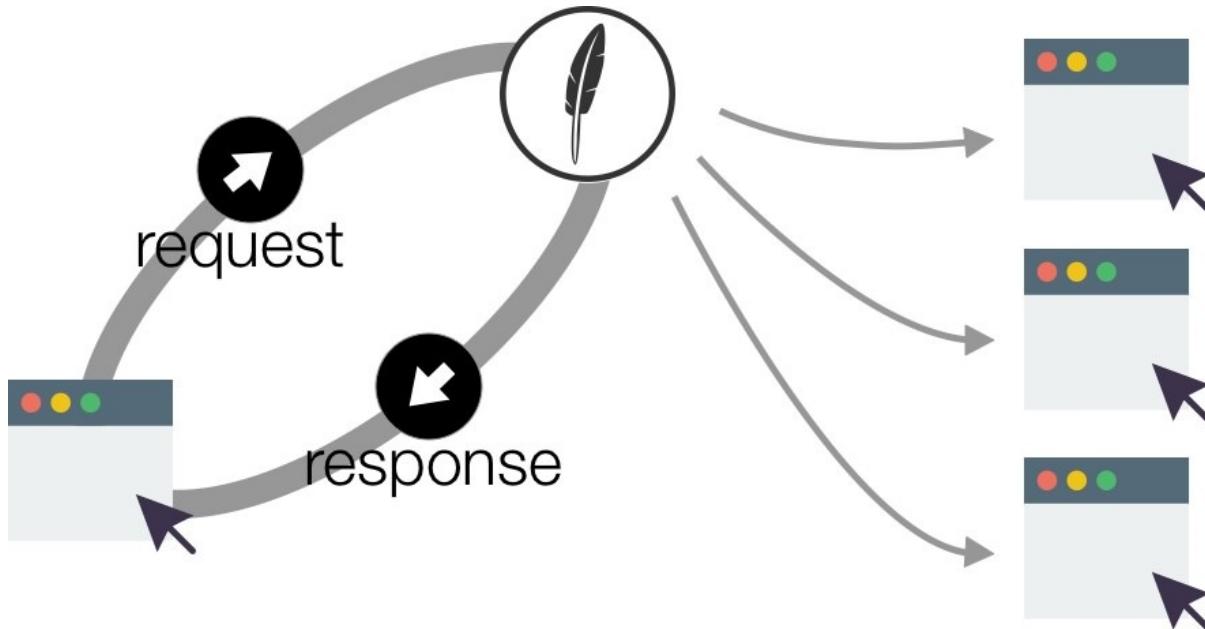
You might not want to use the Feathers NeDB adapter as it may not be opened more than once in a process. You can work around this with mocha's `--require` option, opening it once and attaching it to Nodejs' `global` object to the tests.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Real-time

In Feathers, [real-time](#) means that services automatically send created, updated, patched and removed events when a create, update, patch or remove service method is complete. Clients can listen for these events and then react accordingly.



The client in a chat room app, for example, could automatically receive all messages posted by any of the participants, and then display them. This is **much** simpler than the traditional design pattern which requires long-polling of the server.

As another example, the client could maintain a local copy of part of a database table. It can keep it up to date by listening to events for that table.

Real-time. Real-time events are sent only to Feathers WebSocket clients. They are not sent to Feathers REST nor HTTP REST clients. These would have to implement a traditional long-polling design. **Conclusion:** Use Feathers WebSocket clients.

Let's create an event listener for the [Feathers Websocket Client](#) we already have.

Working example

- Server code: [examples/step/01/websocket/1.js](#)
- Listener code: [common/public/listener.html](#) and [listener-app.js](#)
- Client code: [common/public/socketio.html](#) and [feathers-app.js](#)
- Start the server: `node ./examples/step/01/websocket/1.js`
- Start the listener by pointing a browser tab at localhost:3030/listener.html
- Start making changes by pointing a browser tab at: localhost:3030/socketio.html

Implementing a listener

Implementing the listener [common/public/listener-app.js](#) is straight forward.

```
const users = feathersClient.service('/users');

users.on('created', user => console.log('created', user));
users.on('removed', user => console.log('removed', user));

console.log('Listening for user events.');
```

Filtering

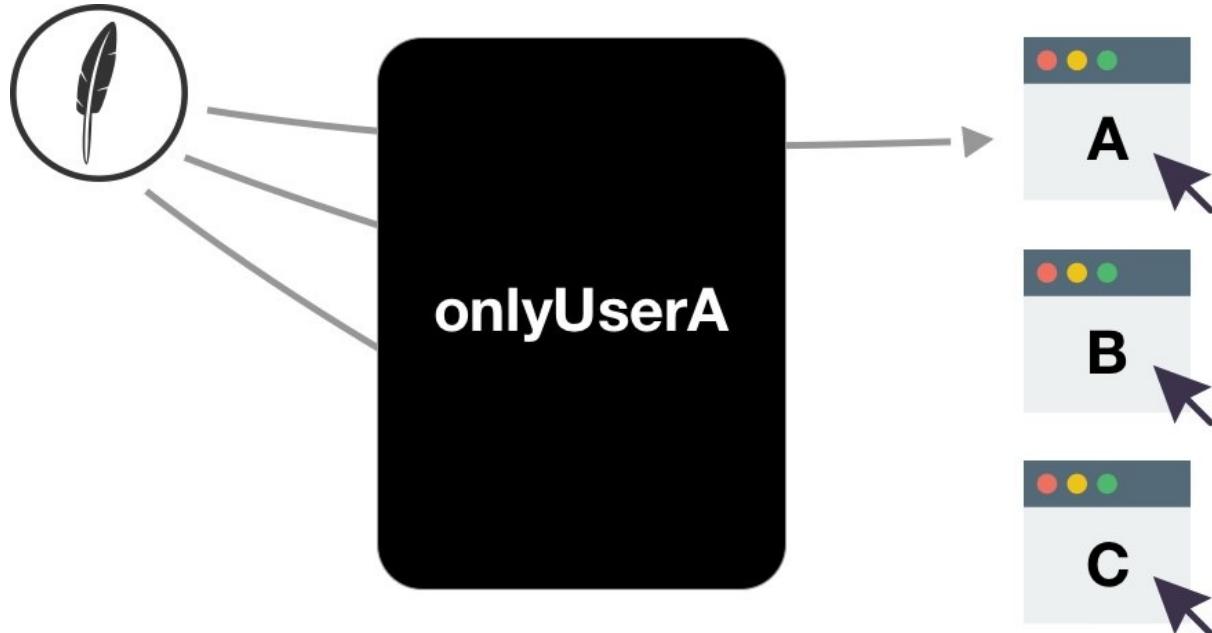
Our listener's console displays:

```
Listening for user events.
created
{email: "jane.doe@gmail.com", password: "11111", role: "admin", _id: "qyRMR6abq8RHV29R"}
created
{email: "john.doe@gmail.com", password: "22222", role: "user", _id: "XI6e3bZcoubp6Hyr"}
created
{email: "judy.doe@gmail.com", password: "33333", role: "user", _id: "qeYSi2KrkwIUMoaE"}
```

You usually wouldn't want to send passwords to clients.

In many cases you probably want to be able to send certain events to certain clients, say maybe only ones that are authenticated.

The server can control what data is sent to which clients with [event filters](#).



For example, we could send `users` events only to authenticated users and remove `password` from the payload by adding this to the server code:

```
const users = app.service('users');
users.filter((data, connection) => {
  delete data.password;
  return connection.user ? data : false;
});
```

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Generators

We've been writing code "by hand" in order to understand how basic Feathers works. We will now start using Feathers generators since we have the background to understand what they produce.

Generators help eliminate boilerplate.

We've seen that Feathers, even when coded "by hand", eliminates the majority of the boilerplate typically in a CRUD project. Generators will eliminate even more.

Generators. Feathers generators produce very little code because Feathers is so succinct. You can easily understand the generated code because its no different from what we've been coding "by hand" so far. Some other frameworks make things "seem" easy by generating thousands of lines of code for you and, in the process, making it almost impossible to implement anything not supported out of the box by their generators.

Generators structure your app.

The generated modules are structured as recommended by the Feathers team.

Generators handle database specifics.

The generators will generate code for different databases so you don't have to investigate how to do so.

Install the generators

You can install the Feathers generators with

```
npm install -g feathers-cli
```

What's next?

Now that we installed the Feathers command line tool we can [generate the application](#).

Is anything wrong, unclear, missing?

[Leave a comment](#).

Generating an app

Now let's write a new project using the Feathers generators.

This project will have users who may be members of one or more teams. We want to display teams with all their members.

Create the app

The first thing we do is generate the basic app. For that, we will first have to create and move into a new folder:

```
mkdir feathers-app  
cd feathers-app
```

Then we can run:

```
feathers generate app
```

```
gen1$ feathers generate app  
? Project name basic-guide-generated-project  
? Description Generated app for Feathers The Basics Guide  
? What folder should the source files live in? src  
? Which package manager are you using (has to be installed globally)? npm  
? What type of API are you making? (Press <space> to select, <a> to toggle all,  
<i> to inverse selection) REST, Realtime via Socket.io  
    create package.json  
    create config/default.json
```

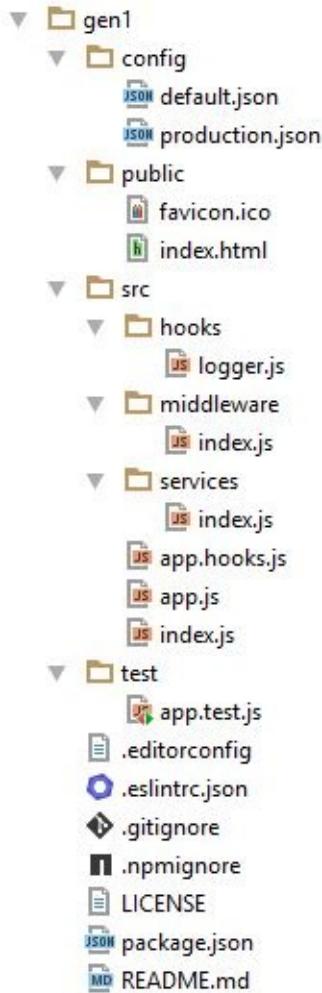
You can start the generated base application using

```
npm start
```

And then go to localhost:3030 to see a standard welcome page.

The generator creates some [modules](#) reflecting your choices. The modules are properly wired together and structured as recommended by the Feathers team.

They are mostly [boilerplate](#) and organized as follows:



config/

Contains the configuration files for the app. `production.json` values override `default.json` ones when in production mode, i.e. when you run `NODE_ENV=production node path/to/your/server.js`.

node_modules/

The generator installs the project dependencies here using either `npm`, or `yarn` if that's installed. The dependencies are enumerated in `package.json`.

public/

Contains the resources to be served. A sample favicon and [HTML file](#) are included.

src/

Contains the Feathers server.

- **hooks/** contains your custom hooks, usually those general enough to be used with multiple services. A simple but useful `logger` is provided as an example.

- **middleware/** contains your Express middleware.
- **services/** will contain the services.
- [index.js](#) is used by node to start the app.
- [app.js](#) configures Feathers and Express.
- [app.hooks.js](#) contains hooks which have to run for **all** services. **We have not covered this capability before.** You can configure such hooks [like this](#).

test/

Contains the tests for the app. [app.test.js](#) tests that the index page appears, as well as 404 errors for HTML pages and JSON.

.editorconfig

is compatible with the [EditorConfig project](#) and helps developers define and maintain consistent coding styles among different editors and IDEs.

.eslintrc.json

contains defaults for limiting your code with [ESLint](#).

.gitignore

specifies [intentionally untracked files](#) which [git](#), [GitHub](#) and other similar projects ignore.

.npmignore

specifies [files](#) which are not to be published for distribution.

LICENSE

contains the [license](#) so that people know how they are permitted to use it, and any restrictions you're placing on it.

It defaults to the Feathers license.

package.json

contains [information](#) which [npm](#), [yarn](#) and other package managers need to install and use your package.

What's next?

The generated code will look familiar. It contains nothing more than what we have covered previously. The main advantages of the Feathers generators are

- Generators structure your app. The generated modules are structured as recommended by the Feathers team.
- Generators write the repetitive boilerplate so you don't have to.
- Generators handle database specifics. The generators will generate code for different databases so you don't have to investigate how to do so.

Next we will [add authentication to the application we just generated](#).

Is anything wrong, unclear, missing?

[Leave a comment](#).

Add authentication

We can now use the generator to add some local authentication to the app by running

```
feathers generate authentication
```

```
gen2$ feathers generate authentication
? What authentication providers do you want to use? Other PassportJS strategies
not in this list can still be configured manually. Username + Password (Local)
? What is the name of the user (entity) service? users
? What kind of service is it? NeDB
? What is the database connection string? nedb://../data
  force config/default.json
  create src/authentication.js
```

The generator will add some new modules and modify some existing ones. You can see all the changes here: [Unified](#) | [Split](#)

New modules

The directory has changed:



The users service

The generator has added a `users` service to the app because local authentication requires we keep a database of users.

This caused the following modules to be added:

- `src/models/users.model.js` describes how `users` is indexed. `Nedb` is a [NoSQL](#) database that's simple to configure.
- `src/services/users` contains the rest of the `users` service.

- `users.service.js` configures the service.
 - `users.hooks.js` configures the hooks for the service. The `authenticate('jwt')` hooks ensure only authenticated users can perform method calls. The `hashPassword()` hook encrypts the password when a new user is added.
 - `users.filters.js` will allow you to control which clients are notified when a user is mutated.
- `test/services/users.test.js` tests that the service gets configured.

The service has to be wired into the app, so the generator made the following changes:

- `config/default.json` now ([Unified](#) | [Split](#)) keeps the path of the NeDB tables.
- `src/services/index.js` now ([Unified](#) | [Split](#)) configures the `users` service.

The authentication service

The generator also added an `authentication` service to the app. Its responsible for authenticating clients against the `users` service, generating JWT tokens and verifying them.

The `authentication` service is a **custom service**, not a database service. It has no model, no database table, no hooks to run when one of its methods is called.

- So instead of creating a set of folders as was done for `users`, the generator creates the only module `authentication` needs as `src/authentication.js`

This service also has to be wired into the app, so the generator made the following change:

- `src/config/default.json` now ([Unified](#) | [Split](#)) retains authentication information.
- `src/app.js` now ([Unified](#) | [Split](#)) configures the `authentication` service.

Other changes

The changes to our app have introduced new dependencies and they need to be defined.

- `package.json` now ([Unified](#) | [Split](#)) records them.

What's next?

We have not previously covered Feathers authentication, so the authentication service written for that is brand new to us. You can refer to the authentication [API](#) and [guides](#) for more details.

A `users` service was created as its needed for the local authentication. That generated code contains no surprises for us as we have covered it before.

Next, we will [generate a new service](#).

Is anything wrong, unclear, missing?

[Leave a comment](#).

Add the teams service

We now generate the teams service using

```
feathers generate service
```

```
gen3$ feathers generate service
? What kind of service is it? NeDB
? What is the name of the service? teams
? Which path should the service be registered on? /teams
? Does the service require authentication? Yes
  create src/services/teams/teams.service.js
  force src/services/index.js
```

The generator will add some new modules and modify some existing ones. You can see all the changes here: [Unified](#) | [Split](#)

New modules

The directory has changed:



The teams service

We saw the `users` service being added previously. The `teams` service has been added in exactly the same way. There is nothing new. The boilerplate differs only in the names of the services.

Generators. The Feathers generators are great for roughing out a project, creating something in its approximate, but not finished, form. The generators will write most of the boilerplate you need, while you concentrate on the unique needs of the project.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Add the populate hook

When we obtain a teams record, we want to add the team's users to the team record. This requires a hook and therefore we generate the scaffolding for a hook using:

```
feathers generate hook
```

```
gen4$ feathers generate hook
? What is the name of the hook? populateTeams
? What kind of hook should it be? after
? What service(s) should this hook be for (select none to add it yourself)?
  teams
? What methods should the hook be for (select none to add it yourself)? find
  create src/hooks/populate-teams.js
  force src/services/teams/teams.hooks.js
```

The generator will add some new modules and modify some existing ones. You can see all the changes here: [Unified](#) | [Split](#)

New modules

The directory has changed:



The populateTeams hook

The generator has roughed out an after hook for the `teams` service. This hook doesn't do anything so far, but it's been placed in the structure and wired into the app.

This caused the following modules to be added:

- [src/hooks/populate-teams.js](#) contains code for a hook that presently does nothing.
- [test/hooks/populate-teams.test.js](#) tests that the hook is configured.

The hook had to be wired into the app, so the generator made the following changes:

- [src/services/teams/teams.hooks.js](#) now ([Unified](#) | [Split](#)) uses the `populateTeams`. We told the generator to create an `after` hook for the `find` method, and that is when it is being run.

ProTip: What you put in `populate-teams.js` is up to you. You'd likely use the `populate` hook for DB adapters other than Sequelize. You may decide to use the more performant internal `populate` features for Sequelize. The generator creates a hook which does nothing.

What's next?

The generated code, once again, contains no surprises for us as we have covered it before. Now we can [run our application](#).

Is anything wrong, unclear, missing?

[Leave a comment](#).

Run the generated application

Now we are good to run the automated tests for our application and start the server.

Run the tests

The generator wrote [some basic tests](#) for what it generated. Let's run them.

```
gen4$ npm run test
> npm run eslint && npm run mocha
> eslint src/. test/. --config .eslintrc.json

> mocha test/ --recursive

You are using the default filter for the users service. For more information about event filters see https://docs.feathersjs.com/api/events.html#event-filtering
You are using the default filter for the teams service. For more information about event filters see https://docs.feathersjs.com/api/events.html#event-filtering

  Feathers application tests
    ✓ starts and shows the index page (207ms)
    404
      ✓ shows a 404 HTML page (97ms)
      ✓ shows a 404 JSON error without stack trace (54ms)

  'populateTeams' hook
    ✓ runs the hook

  'teams' service
    ✓ registered the service

  'users' service
    ✓ registered the service

  6 passing (464ms)

gen4$
```

npm run test runs the `test` script in [package.json](#).

```
"scripts": {
  "test": "npm run eslint && npm run mocha",
  "eslint": "eslint src/. test/. --config .eslintrc.json",
  "start": "node src/",
  "mocha": "mocha test/ --recursive"
}
```

First [ESLint](#) runs, using the options in [.eslintrc.json](#).

ESLint checks the syntax and basic coding patterns of the modules in `src/` and `test/`. Any informative messages would be logged to the console and the processing terminated.

Next the tests themselves are run. They were written in [Mocha](#) and use Mocha's default options. Each test logs to the console as it runs. The summary shows that 6 tests were successful and there were no failures.

So we can now be sure that:

- The generated code follows established best-practices for syntax and basic coding patterns.
- The generated code is wired together properly.

Start the server

Since everything looks OK, let's start the server.

```
gen4$ npm run start
> node src/
You are using the default filter for the users service. For more information about event filters see https://docs.feathersjs.com/api/events.html#event-filtering
You are using the default filter for the teams service. For more information about event filters see https://docs.feathersjs.com/api/events.html#event-filtering
info: Feathers application started on localhost:3030
```

The `info` line indicates the server for our roughed out app started properly.

The `You are using the default filter for ...` lines are interesting. They are logged from [here](#) and [here](#).

Feathers [real-time events](#) will notify all WebSocket clients of mutations occurring in Feathers DB services. You may want to [filter](#) who gets to see which events.

These messages are logged just to remind you to do so.

About the config files

We changed to the generated app's directory (`gen4`) to start the server. Its nice to use `npm run start` as then we don't need to know where the server starting code resides. However we could just as easily have run `node ./src`.

One thing the generated code assumes is that the `config` [directory](#) is located in the current directory. So we wouldn't be able to start the server with `node path/to/app/src` because the config files wouldn't be found.

You can get around this by explicitly providing the direct or relative path of the configuration directory:

```
# Linux, Mac
NODE_CONFIG_DIR=path/to/app/config node path/to/app/src
# Windows
SET NODE_CONFIG_DIR=path/to/app/config
node path/to/app/src
```

What's next?

The Feathers generators are great for roughing out a project, creating something in its approximate, but not finished, form.

Generators. You can also use them later on to add additional services and hooks as your app evolves.

We now have the boilerplate for our app, and we start adding the custom code it requires.

We've learned how the generator works, and we understand the code it produces. Let's now use this knowledge to build a [Chat Application](#).

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Creating a Chat Application

Well alright! Let's build our first Feathers app! We're going to build a real-time chat app with **NeDB** as the database. It's a great way to cover all the things that you'd need to do in a real world application and how Feathers can help. It also makes a lot more sense to have a real-time component than a Todo list.

The screenshot shows a Feathers real-time chat interface. At the top, it says "Feathers CHAT". On the left, there's a sidebar with a "3 users" badge and a list of users: daff@neyeon.com, cory.m.smith@gm, and e.kryski@gmail.co. The main area shows a message from cory.m.smith@gmail.com dated Mar 14th, 11:48:23, which reads "Hey guys!". Below the message input field, there are "Sign Out" and "Send" buttons.

In this tutorial you go from nothing to a real-time chat app complete with signup, login, token based authentication and authorization all with a RESTful and real-time API.

You can find a complete working example [here](#).

Creating the application

Create a new application using the generator.

Generating a service

Add an API endpoint to store messages.

Building a frontend

Learn how to use Feathers in the browser by creating a small real-time chat frontend.

Adding Authentication

Add user registration and login.

Processing data

Add and sanitize data.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Creating the application

In this part we are going to create a new Feathers application using the generator.

Generating the application

With everything [set up](#) let's create a directory for our new app:

```
$ mkdir feathers-chat
$ cd feathers-chat/
```

Now we can generate the application:

```
$ feathers generate app
```

When presented with the project name just hit enter, or enter a name (no spaces).

Next, enter in a short description of your app.

The next prompt asking for the source folder can be answered by just hitting enter. This will put all source files into the `src/` folder.

The next prompt will ask for the package manager you want to use. The default is the standard [npm](#).

Note: Choosing [Yarn](#) will make for faster installation times but requires Yarn installed globally via `npm install yarn -g` first.

You're now presented with the option to choose which transport you want to support. Since we're setting up a real-time and REST API we'll go with the default REST and Socket.io options. So just hit enter.

Once you confirm the final prompt you will see something like this:

```
[daffl:~/feathers-chat $ feathers generate app
? Project name feathers-chat
? Description A Feathers chat application
? What folder should the source files live in? src
? Which package manager are you using (has to be installed globally)? npm
? What type of API are you making? (Press <space> to select, <a> to toggle all, <i> to inverse selection)
REST, Realtime via Socket.io
  create package.json
  create config/default.json
  create LICENSE
  create public/favicon.ico
  create public/index.html
  create test/app.test.js
  create .editorconfig
  create .eslintrc.json
  create .npmignore
  create src/app.hooks.js
  create src/hooks/logger.js
  create src/index.js
  create src/middleware/index.js
  create src/services/index.js
  create .gitignore
  create README.md
  create src/app.js
  create config/production.json
```

The structure and purpose of all those files that have just been created are covered in the [generator chapter](#).

Running the server and tests

The server can now be started by running

```
npm start
```

After that, you can see a welcome page at localhost:3030. When making modifications, remember to stop (CTRL + C) and start the server again.

The app also comes with a set of basic tests which can be run with

```
npm test
```

What's next?

We scaffolded a new Feathers application. The next step is to [create a service for messages](#).

Is anything wrong, unclear, missing?

[Leave a comment](#).

Creating a service

Now that we have our [Feathers application generated](#) we can create a new API endpoint to store messages.

Generating a service

In Feathers any API endpoint is represented as a [service](#) which we already learned about in the [basics guide](#). To generate a new service we can run

```
feathers generate service
```

First we have to choose what kind of service we would like to create. You can choose between many databases and ORMs but for this guide we will just go with the default [NeDB](#). NeDB is a database that stores its data locally in a file and requires no additional configuration or a database server running.

Next we are asked for the name of the service which we can answer with `messages` and then can answer the next question for the path with the default (`/messages`) by pressing enter.

The database connection string (in the case of NeDB the name of the path where it should store its database files) can also be answered with the default.

Confirming the last prompt will create a couple of files and wire our service up:

```
[daffl:~/feathers-chat $ feathers generate service
? What kind of service is it? NeDB
? What is the name of the service? messages
? Which path should the service be registered on? /messages
? What is the database connection string? nedb://../data
  force config/default.json
  create src/services/messages/messages.service.js
  force src/services/index.js
  create src/models/messages.model.js
  create src/services/messages/messages.hooks.js
  create src/services/messages/messages.filters.js
  create test/services/messages.test.js
```

Et voilà! We have a fully functional REST and real-time API for our messages.

Testing the API

If we now start our API with

```
npm start
```

We can go to `localhost:3030/messages` and will see an (empty) response from our new messages service.

We can also `POST` new messages and `PUT`, `PATCH` and `DELETE` existing messages (via `/messages/<_id>`), for example from the command line using [CURL](#):

```
curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --data-binary '{ "name": "Curler", "text": "Hello from the command line!" }'
```

Or with a REST client, e.g. [Postman](#) using this button:

[!\[\]\(14abeea2bc9e768a1eb56545bc1ea001_img.jpg\) Run in Postman](#)

If we now go to localhost:3030/messages again we will see the newly created message(s).

What's next?

With just one command, we created a fully functional REST and real-time API endpoint. Before we dive into authentication and processing data, [let's create a simple web application](#) that uses our new chat message endpoint.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Building a frontend

In this chapter we will create a very simple web application for the messages service [that we just created](#). We won't be using a framework like jQuery, Angular, React or VueJS (for more information about those, see the [frameworks](#) section). Instead we will go with plain old JavaScript that will work in any modern browser (latest Chrome, Firefox and the Edge Internet Explorer).

Using Feathers on the client

We could use a REST client (making AJAX request) or websockets messages directly (both of which are totally possible with Feathers), but instead we will leverage one of the best features of Feathers, namely that it works just the same as a client in the browser, with React Native or on other NodeJS servers.

Our `public/` folder already has an `index.html` page that currently shows a generated homepage if you go to localhost:3030 in the browser. We will modify that page to show our chat messages and a form to send new ones.

First, let's add the browser version of Feathers to the page. We can do so by linking to a CDN that has the [client browser build](#) of Feathers. Update `public/index.html` to look like this:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1, user-scalable=0" />
    <title>Feathers Chat</title>
    <link rel="shortcut icon" href="favicon.ico">
  </head>
  <body>
    <div id="app" class="flex flex-column"></div>
    <script src="//unpkg.com/feathers-client@^2.0.0-pre.1/dist/feathers.js">
    </script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

Connecting to the API

Now we can create `public/app.js` with the following Feathers client setup:

```
const socket = io();
const client = feathers();
client.configure(feathers.hooks());

// Create the Feathers application with a `socketio` connection
client.configure(feathers.socketio(socket));

// Get the service for our `messages` endpoint
const messages = client.service('messages');

// Log when anyone creates a new message in real-time!
messages.on('created', message =>
  alert(`New message from ${message.name}: ${message.text}`)
);

// Create a test message
messages.create({
```

```

    name: 'Test user',
    text: 'Hello world!'
});

messages.find().then(page => console.log('Current messages are', page));

```

This will connect to our API server using Socket.io, send a test message and also listen to any new message in real-time showing it in an alert window when going to the page at localhost:3030. Once you have created the message you can also see that it showed up in the localhost:3030/messages endpoint.

Note: The `feathers` namespace is added by the browser build and `io` is available through the `socket.io/socket.io.js` script. For more information on using Feathers in the browser and with a module loader like Webpack or Browserify see the [client chapter](#).

Sending and displaying messages

Alright. We can create and listen to new messages and also list all messages. All that is left to do now is create some HTML from the message list and a form to create new messages. Let's update `public/index.html` with some HTML for our chat to look like this:

```

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1, user-scalable=0" />
    <title>Feathers Chat</title>
    <link rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.2.0/public/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.2.0/public/chat.css">
  </head>
  <body>
    <div id="app" class="flex flex-column">
      <main class="flex flex-column">
        <header class="title-bar flex flex-row flex-center">
          <div class="title-wrapper block center-element">
            
            <span class="title">Chat</span>
          </div>
        </header>

        <div class="flex flex-row flex-1 clear">
          <div class="flex flex-column col col-12">
            <main class="chat flex flex-column flex-1 clear"></main>

            <form class="flex flex-row flex-space-between" id="send-message">
              <input type="text" placeholder="Your name" name="name" class="col col-3">
              <input type="text" placeholder="Enter your message" name="text" class="col col-7">
              <button class="button-primary col col-2" type="submit">Send</button>
            </form>
          </div>
        </main>
      </div>
      <script src="//unpkg.com/feathers-client@^2.0.0-pre.1/dist/feathers.js">
      </script>
      <script src="/socket.io/socket.io.js"></script>
      <script src="app.js"></script>
    </body>
  </html>

```

Then we can update `public/app.js` with the functionality to get, show and send messages like this:

```

const socket = io();
const client = feathers();

// Create the Feathers application with a `socketio` connection
client.configure(feathers.socketio(socket));

// Get the service for our `messages` endpoint
const messages = client.service('messages');

// Add a new message to the list
function addMessage(message) {
  const chat = document.querySelector('.chat');

  chat.insertAdjacentHTML('beforeend', `<div class="message flex flex-row">
    
    <div class="message-wrapper">
      <p class="message-header">
        <span class="username font-600">${message.name}</span>
      </p>
      <p class="message-content font-300">${message.text}</p>
    </div>
  </div>`);

  chat.scrollTop = chat.scrollHeight - chat.clientHeight;
}

messages.find().then(page => page.data.forEach(addMessage));
messages.on('created', addMessage);

document.getElementById('send-message').addEventListener('submit', function(ev) {
  const nameInput = document.querySelector('[name="name"]');
  // This is the message text input field
  const textInput = document.querySelector('[name="text"]');

  // Create a new message and then clear the input field
  client.service('messages').create({
    text: textInput.value,
    name: nameInput.value
  }).then(() => {
    textInput.value = '';
  });
  ev.preventDefault();
});

```

If you now open localhost:3030 you can see an input field for your name and the message which will show up in other browsers in real-time.

What's next?

In this chapter we looked at how to use Feathers on the client and created a simple real-time chat application frontend to show and send messages. In the next chapters we will move back to the server and add [authentication](#) and learn about [processing data](#).

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Adding authentication

In the previous chapters we [created our Feathers chat application](#) and [initialized a service](#) for storing messages. We also build a simple [real-time frontend for the browser](#). However, for a proper chat application we need to be able to register and authenticate users.

Generating authentication

To add authentication to our application we can run

```
feathers generate authentication
```

This will first ask us which authentication providers we would like to use. In this guide we will only cover local authentication which is already selected so we can just confirm by pressing enter.

Next we have to define the service we would like to use to store user information. Here we can just confirm the default `users` and the database with the default NeDB:

```
daffl:~/feathers-chat $ feathers generate authentication
[daffl:~/feathers-chat $ feathers generate authentication
? What authentication providers do you want to use? Other PassportJS strategies not in this list can still be configured manually. Username + Password (Local)
[?] What is the name of the user (entity) service? users
[?] What kind of service is it? NeDB
  force config/default.json
  create src/authentication.js
  force src/app.js
  create src/services/users/users.service.js
  force src/services/index.js
  create src/models/users.model.js
  create src/services/users/users.hooks.js
  create src/services/users/users.filters.js
  create test/services/users.test.js
```

Creating a user and logging in

We just created a `users` service and enabled local authentication. When restarting the application we can now create a new user with `email` and `password` similar to what we did with messages and then use the login information to get a JWT (for more information see the [How JWT works guide](#)).

Creating the user

We will create a new user with the following data:

```
{
  "email": "feathers@example.com",
  "password": "secret"
}
```

The generated user service will automatically securely hash the password in the database for us and also exclude it from the response (passwords should never be transmitted). There are several ways to create a new user, for example via CURL like this:

```
curl 'http://localhost:3030/users/' -H 'Content-Type: application/json' --data-binary '{ "email": "feathers@example.com", "password": "secret" }'
```

With a REST client, e.g. [Postman](#) using this button:

[▶ Run in Postman](#)

Or via the client we used in the [frontend chapter](#) by adding the following to `public/app.js`:

```
// Create a test new user
client.service('users').create({
  email: 'feathers@example.com',
  password: 'secret'
});
```

Note: Creating a user with the same email address will only work once and fail when it already exists in the database.

Getting a token

To create a JWT we can now post the login information with the strategy we want to use (`local`) to the `authentication` service:

```
{
  "strategy": "local",
  "email": "feathers@example.com",
  "password": "secret"
}
```

Via CURL:

```
curl 'http://localhost:3030/authentication/' -H 'Content-Type: application/json' --data-binary '{ "strategy": "local", "email": "feathers@example.com", "password": "secret" }'
```

With a REST client, e.g. [Postman](#):

[▶ Run in Postman](#)

The returned token can now be used to authenticate the user it was created for by adding it to the `Authorization` header of new HTTP requests.

The Feathers client from the [frontend chapter](#) already has authentication (and storing the generated token in `LocalStorage`) built in and can be used by adding this to `public/app.js`:

```
client.configure(feathers.authentication({
  storage: window.localStorage
}));

client.authenticate({
  strategy: 'local',
  email: 'feathers@example.com',
  password: 'secret'
}).then(token => {
  console.log('User is logged in');
});
```

Then we can update `public/app.js` to look like this:

```
const socket = io();
const client = feathers();
```

```
// Create the Feathers application with a `socketio` connection
client.configure(feathers.socketio(socket));

// Get the service for our `messages` endpoint
const messages = client.service('messages');

// Configure authentication
client.configure(feathers.authentication({
  storage: window.localStorage
}));

client.authenticate({
  strategy: 'local',
  email: 'feathers@example.com',
  password: 'secret'
}).then((token) => {
  console.log('User is logged in', token);

  // At this point we have a valid token, so we can fetch restricted data.
  messages.find().then(page => page.data.forEach(addMessage));
  messages.on('created', addMessage);
});

// Add a new message to the list
function addMessage(message) {
  const chat = document.querySelector('.chat');

  chat.insertAdjacentHTML('beforeend', `<div class="message flex flex-row">
    
    <div class="message-wrapper">
      <p class="message-header">
        <span class="username font-600">${message.name}</span>
      </p>
      <p class="message-content font-300">${message.text}</p>
    </div>
  </div>`);

  chat.scrollTop = chat.scrollHeight - chat.clientHeight;
}

document.getElementById('send-message').addEventListener('submit', function(ev) {
  const nameInput = document.querySelector('[name="name"]');
  // This is the message text input field
  const textInput = document.querySelector('[name="text"]');

  // Create a new message and then clear the input field
  client.service('messages').create({
    text: textInput.value,
    name: nameInput.value
  }).then(() => {
    textInput.value = '';
  });
  ev.preventDefault();
});
```

Securing the messages service

Now we have to restrict our messages service to authenticated users. If we run `feathers generate authentication` before generating other services it will ask if the service should be restricted to authenticated users. Because we created the messages service first, however we have to update `src/services/messages/messages.hooks.js` manually to look like this:

```
const { authenticate } = require('feathers-authentication').hooks;
```

```
module.exports = {
  before: {
    all: [ authenticate('jwt') ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
  after: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
  error: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  }
};
```

This will now only allow users with a valid JWT to access the service.

What's next?

In this chapter we initialized authentication and created a user and JWT. We can now use that user information to [process new message data](#).

Is anything wrong, unclear, missing?

[Leave a comment](#).

Processing data

Now that we can [create and authenticate users](#), we are going to process data, sanitize the input we get from the client and add additional information.

Sanitizing new message

When creating a new message, we automatically want to sanitize HTML input, add the user that sent it and include the date the message has been created before saving it in the database. This is where hooks come into play, in our case specifically a *before* hook. To create a new hook we can run:

```
feathers generate hook
```

The hook we want to create will be called `process-message`. Since we want to pre-process our data, the next prompt asking for what kind of hook, we will choose `before` from the list.

Next we will see a list of all our services we can add this hook to. For this hook we will only choose the `messages` service (navigate to the entry with the arrow keys and select it with the space key).

A hook can run before any number of [service methods](#), for this one we will only select `create`. After confirming the last prompt we will see something like this:

```
[daffl:~/feathers-chat $ feathers generate hook
? What is the name of the hook? process-message
? What kind of hook should it be? before
? What service(s) should this hook be for (select none to add it yourself)?
  messages
? What methods should the hook be for (select none to add it yourself)? create
  create src/hooks/process-message.js
  force src/services/messages/messages.hooks.js
  create test/hooks/process-message.test.js
```

This will create our hook and wire it up to the service we selected. Now it is time to add some code. Update `src/hooks/process-message.js` to look like this:

```
'use strict';

// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/api/hooks.html

module.exports = function() {
  return function(hook) {
    // The authenticated user
    const user = hook.params.user;
    // The actual message text
    const text = hook.data.text
      // Messages can't be longer than 400 characters
      .substring(0, 400)
      // Do some basic HTML escaping
      .replace(/&/g, '&').replace(/</g, '<').replace(/>/g, '>');
    // Override the original data
    hook.data = {
      text,
      // Set the user id
      userId: user._id,
      // Add the current time via `getTime`
      createdAt: new Date().getTime()
    }
  }
}
```

```

    };

    // Hooks can either return nothing or a promise
    // that resolves with the `hook` object for asynchronous operations
    return Promise.resolve(hook);
};

};

```

This will do several things:

1. Truncate the messages `text` property to 400 characters and do some basic HTML escaping.
2. Update the data submitted to the database to contain
 - o The new truncated and sanitized text
 - o The currently authenticated user (so we always know who sent it)
 - o The current (creation) date
3. Return a Promise that resolves with the hook object (this is what any hook should return)

Adding a user avatar

Let's create one more hook that adds a link to the [Gravatar](#) image of the users email address so we can show an avatar. After running

```
feathers generate hook
```

The selections are almost the same as our previous hook:

- The hook will be called `gravatar`
- It will be a `before` hook
- On the `users` service
- For the `create` method

```
[daffl:~/feathers-chat $ feathers generate hook
? What is the name of the hook? gravatar
? What kind of hook should it be? before
? What service(s) should this hook be for (select none to add it yourself)?
  users
? What methods should the hook be for (select none to add it yourself)? create
  create src/hooks/gravatar.js
    force src/services/users/users.hooks.js
  create test/hooks/gravatar.test.js
```

Then we update `src/hooks/gravatar.js` with the following code:

```
'use strict';

// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/api/hooks.html

// We need this to create the MD5 hash
const crypto = require('crypto');

// The Gravatar image service
const gravatarUrl = 'https://s.gravatar.com/avatar';
// The size query. Our chat needs 60px images
const query = 's=60';

module.exports = function() {
  return function(hook) {
    // The user email
    const { email } = hook.data;
```

```
// Gravatar uses MD5 hashes from an email address to get the image
const hash = crypto.createHash('md5').update(email).digest('hex');

hook.data.avatar = `${gravatarUrl}/${hash}?${query}`;

// Hooks can either return nothing or a promise
// that resolves with the `hook` object for asynchronous operations
return Promise.resolve(hook);
};

};
```

Here we use [Node's crypto library](#) to create an MD5 hash of the users email address. This is what Gravatar uses as the URL for the avatar of an email address. If we now create a new user it will add the link to the image in the `gravatar` property.

Populating the message sender

In the `process-message` hook we are currently just storing the users `_id` in the message. We want to show more than the `_id` in the UI, so we'll need to populate more data in the message response. In order to show the right user information we want to include that information in our messages.

We could do this by creating our own hook but adding related entities is quite common and already implemented in the [populate common hook](#). In order to use the hook we have to update the `src/services/messages/messages.hooks.js` file to look like this:

```
'use strict';

const { authenticate } = require('feathers-authentication').hooks;
const { populate } = require('feathers-hooks-common');
const processMessage = require('../hooks/process-message');

module.exports = {
  before: {
    all: [ authenticate('jwt') ],
    find: [],
    get: [],
    create: [ processMessage() ],
    update: [ processMessage() ],
    patch: [ processMessage() ],
    remove: []
  },
  after: {
    all: [
      populate({
        schema: {
          include: [
            { service: 'users',
              nameAs: 'user',
              parentField: 'userId',
              childField: '_id'
            }
          ]
        }
      })
    ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
};
```

```
error: {
  all: [],
  find: [],
  get: [],
  create: [],
  update: [],
  patch: [],
  remove: []
}
};
```

This will include the `user` property using the `userId`, retrieving it from the `users` service to all messages.

You can learn more about how the `populate` hook works by checking out the API docs for [feathers-hooks-common](#).

What's next?

In this section we added three hooks to pre- and postprocess our message and user data. We now have a complete API to send and retrieve messages including authentication.

See the [frameworks section](#) for more resources on specific frameworks like React, React Native, Angular or VueJS. You'll find guides for creating a complete chat frontend with signup, logging, user listing and messages. There are also links to full example chat applications built with some popular frontend frameworks.

You can also browse the [API](#) which has a lot of information on the usage of Feathers and its database adaptors.

Is anything wrong, unclear, missing?

[Leave a comment.](#)

Integrating with Frontend Frameworks

Feathers Chat Applications

These guides show how to integrate the [Feathers Chat application](#) with various frontend frameworks.

Full Feathers Chat Examples

While we're working on the guides, you can check out example applications in the [feathers-chat](#) and [feathers-chat-vuex](#) repos.

Feathers Chat - React

Guide not yet published on the blog. See the full example app [here](#)

Feathers Chat - Vue.js 2

Guide not yet published on the blog.

Feathers Chat - Vue.js 2 with `feathers-vuex`

Guide not yet published on the blog. See the full example app [here](#)

Vue.js

[Integrating Nuxt into your Feathers Application](#)

Learn how to integrate the Nuxt server-side rendering framework for Vue.js into your Feathers application.

Authentication Guides & Recipes

How JWT Works

Learn more about JWT and how it might differ from authentication methods you've used, previously. (This guide is a work in progress.)

What's new in `feathers-authentication@1.x`

The new `feathers-authentication` introduces a lot of changes. See what's new.

Migrating to `feathers-authentication@1.x`

See what needs to change to upgrade your existing Feathers application from `feathers-authentication@0.7.x`.

Auth Recipe: Customize the JWT Payload

You can customize the JWT payload. Learn important security implications before you decide to do it.

Auth Recipe: Customize the Login Response

Learn how you can customize the response after a user has attempted to login.

Auth Recipe: Create Endpoints with Mixed Auth

Learn how to setup an endpoint so that it handles unauthenticated and authenticated users with different responses for each.

Auth Recipe: Basic OAuth

Learn how OAuth (Facebook, Google, GitHub) login works, and how you can use it in your application.

How JSON Web Tokens Work

This guide is a work in progress. There's some useful information here while we make it more user friendly in the context of Feathers. In the meantime, here are a couple of resources on JWT to get more familiar with how it works, in general:

- [The Auth0 JWT Documentation](#) - If you want a good high-level overview.
- [The IETF JWT Specification](#) - If you want to get really technical.

Customizing the JWT Claims

`feathers-authentication@1.x` allows you to customize the data stored inside the JWT. We refer to the data in the JWT as the `payload`. There are a few reserved attributes, which in the [Official JWT Spec](#) are called `claims`. You can customize some of these claims in the [JWT config options on the server](#). To get more familiar with the purpose of each `claim`, please refer to [Section 4 of the Official JWT Specification](#).

FeathersJS Auth Recipe: Customizing the Login Response

The Auk release of FeathersJS includes a powerful new [authentication suite](#) built on top of [PassportJS](#). The new plugins are very flexible, allowing you to customize nearly everything. This flexibility required making some changes. In this guide, we'll look at the changes to the login response and how you can customize it.

Changes to the Login Response

The previous version of `feathers-authentication` always returned the same response. It looked something like this:

```
{  
  token: '<the jwt token>',  
  user: {  
    id: 1,  
    email: 'my@email.com'  
  }  
}
```

The JWT also contained a payload which held an `id` property representing the user `id`. We found that this was too restrictive for some of our more technical apps. For instance, what if you wanted to authenticate a device instead of a user? Or what if you want to authenticate both a device **and** a user? The old plugin couldn't handle those situations. The new one does. To make it work, we started by simplifying the response. The default response now looks like this:

```
{  
  accessToken: '<the jwt token>'  
}
```

The JWT also contains a payload which has a `userId` property.

Based on the above, you can see that we still authenticate a `user` by default. In this case, the `user` is what we call the `entity`. It's the generic name of what is being authenticated. It's customizable, but that's not covered in this guide. Instead, let's focus on what it takes to add the user in the login response.

Customizing the Login Response

The `/authentication` endpoint is now a Feathers service. It uses the `create` method for login and the `remove` method for logout. Just like with all Feathers services, you can customize the response with the [hook API](#). For what we want to do, the important part is the `hook.result`, which becomes the response body. We can use an `after` hook to customize the `hook.result` to return anything that we want:

```
app.service('/authentication').hooks({  
  after: {  
    create: [  
      hook => {  
        hook.result.foo = 'bar';  
      }  
    ]  
  }  
});
```

After a successful login, the `hook.result` already contains the `accessToken`. The above example modified the response to look like this:

```
{  
  accessToken: '<the jwt token>',  
  foo: 'bar'  
}
```

Accessing the User Entity

Let's see how to include the `user` in the response, as was done in previous versions. The `/authentication` service modifies the `hook.params` object to contain the entity object (in this case, the `user`). With that information, you might have already figured out how to get the user into the response. It just has to be copied from `hook.params.user` to the `hook.result.user`:

```
app.service('/authentication').hooks({  
  after: {  
    create: [  
      hook => {  
        hook.result.user = hook.params.user;  
  
        // Don't expose sensitive information.  
        delete hook.result.user.password;  
      }  
    ]  
  }  
});
```

At this point, the response now includes the `accessToken` and the `user`. Now the client won't have to make an additional request for the `user` data. *As is shown in the above example, be sure to not expose any sensitive information.*

Wrapping Up

You've now learned some of the differences in the new `feathers-authentication` plugin. Instead of using two endpoints, it's using a single service. It also has a simplified response, compared to before. Now, you can customize the response to include whatever information you need.

FeathersJS Auth Recipe: Customizing the JWT Payload

The Auk release of FeathersJS includes a powerful new [authentication suite](#) built on top of [PassportJS](#). The new plugins are very flexible, allowing you to customize nearly everything. One feature added in the latest release is the ability to customize the JWT payload using hooks. Let's take a look at what this means, how to make it work, and learn about the potential pitfalls you may encounter by using it.

The JWT Payload

If you read the resources on [how JWT works](#), you'll know that a JWT is an encoded string that can contain a payload. For a quick example, check out the Debugger on [jwt.io](#). The purple section on [jwt.io](#) is the payload. You'll also notice that you can put arbitrary data in the payload. The payload data gets encoded as the section section of the JWT string.

The default JWT payload contains the following claims:

```
const decode = require('jwt-decode')
// Retrieve the token from wherever you've stored it.
const jwt = window.localStorage.getItem('feathers-jwt')
const payload = decode(jwt)

payload === {
  aud: 'https://yourdomain.com', // audience
  exp: 23852348347, // expires at time
  iat: 23852132232, // issued at time
  iss: 'feathers', // issuer
  sub: 'anonymous', // subject
  userId: 1 // the user's id
}
```

Notice that the payload ***is encoded*** and ***IS NOT ENCRYPTED***. It's an important difference. It means that you want to be careful what you store in the JWT payload.

Customizing the Payload with Hooks

The authentication services uses the `params.payload` object in the hook context for the JWT payload. This means you can customize the JWT by adding a `before` hook after the `authenticate` hook.

```
app.service('authentication').hooks({
  before: [
    create: [
      authentication.hooks.authenticate(config.strategies),

      // This hook adds the `test` attribute to the JWT payload by
      // modifying params.payload.
      hook => {
        // make sure params.payload exists
        hook.params.payload = hook.params.payload || {}
        // merge in a `test` property
        Object.assign(hook.params.payload, {test: 'test'})
      }
    ],
    remove: [
      authentication.hooks.authenticate('jwt')
    ]
  ]
})
```

```
    ]
  }
})
```

Now the payload will contain the `test` attribute:

```
const decode = require('jwt-decode')
// Retrieve the token from wherever you've stored it.
const jwt = window.localStorage.getItem('feathers-jwt')
const payload = decode(jwt)

payload === {
  aud: 'https://yourdomain.com',
  exp: 23852348347,
  iat: 23852132232,
  iss: 'feathers',
  sub: 'anonymous',
  userId: 1
  test: true // Here's the new claim we just added
}
```

Important Security Information

As you add data to the JWT payload the token size gets larger. Try it out on [jwt.io](#) to see for yourself. There is an important security issue to keep in mind when customizing the payload. This issue involves the default `HS256` algorithm used to sign the token.

With `HS256`, there is a relationship between the length of the secret (which must be a minimum of 256-bits) and the length of the encoded token (which varies with the payload). A larger secret-to-payload ratio (so the secret is larger than the JWT) will result in a more secure JWT. This also means that keeping the secret size the same and increasing the payload size will actually make your JWT comparatively less secure.

The Feathers generator creates a 2048-bit secret, by default, so there is a small amount of allowable space for putting additional attributes in the JWT payload. It's very important to keep the secret-to-payload length ratio as high as possible to avoid brute force attacks. In a brute force attack, the attacker attempts to retrieve the secret by guessing the secret over and over until getting it right. If your secret is compromised, they will be able to create signed JWT with whatever payload they wish. In short, be cautious about what you put in your JWT payload.

Finally, remember that the secret created by the generator is meant for development purposes, only. You never want to check your **production** secret into your version control system (Git, etc.). It is best to put your production secret in an environment variable and reference it in the app configuration.

FeathersJS Auth Recipe: Create Endpoints with Mixed Auth

The Auk release of FeathersJS includes a powerful new [authentication suite](#) built on top of [PassportJS](#). It can be customized to handle almost any app's authentication requirements. In this guide, we'll look at how to handle a fairly common auth scenario: Sometimes an endpoint needs to serve different information depending on whether the user is authenticated. An unauthenticated user might only see public records. An authenticated user might be able to see additional records.

Setup the Authentication Endpoint

To get started, we need a working authentication setup. Below is a default configuration and `authentication.js`. They contain the same code generated by the [feathers-cli](#). You can create the below setup using the following terminal commands:

1. `npm install -g feathers-cli@latest`
or
`yarn global feathers-cli@latest`
2. `mkdir feathers-demo-local; cd feathers-demo-local`
or a folder name you prefer.
3. `feathers generate app`
use the default prompts.
4. `feathers generate authentication`
 - o Select `Username + Password (Local)` when prompted for a provider.
 - o Select the defaults for the remaining prompts.

config/default.json:

```
{
  "host": "localhost",
  "port": 3030,
  "public": "../public/",
  "paginate": {
    "default": 10,
    "max": 50
  },
  "authentication": {
    "secret": "99294186737032fedad37dc2e847912e1b9393f44a28101c986f6ba8b8bc0eaab48b5b4c5178f55164973c76f8f98f25
23b860674f01c16a23239a2e7d7790ae9fa00b2de5cc0565e335c6f05f2e17fbbee2e8ea0e82402959f1d58b2b2dc5272d09e0c1edf1d364
e9911ecad8172bcd2d41381c9ab319de4979c243925c49165a9914471be0aa647896e981da5aec6801a6dccd1511da11b696d4f6cce3a45
34dab9368661458a466661b1e12170ad21a4045ce1358138caf099fb19e05532336b5626aa376bc158cf84c6a7e0e3dbbb3af666267c08
de12217c9b55aea501e5c36011779ee9dd2e061d0523ddf71cb1d68f83ea5bb1299ca06003b77f0fc69",
    "strategies": [
      "jwt",
      "local"
    ],
    "path": "/authentication",
    "service": "users",
    "jwt": {
      "header": {
        "type": "access"
      },
      "audience": "https://yourdomain.com",
      "subject": "anonymous",
      "issuer": "feathers",
      "algorithm": "HS256",
      "ttl": 60
    }
  }
}
```

```

    "expiresIn": "1d"
},
"local": {
  "entity": "user",
  "service": "users",
  "usernameField": "email",
  "passwordField": "password"
}
},
"nedb": "../data"
}

```

src/authentication.js:

```

'use strict';

const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const local = require('feathers-authentication-local');

module.exports = function () {
  const app = this;
  const config = app.get('authentication');

  app.configure(authentication(config));
  app.configure(jwt());
  app.configure(local(config.local));

  app.service('authentication').hooks({
    before: [
      create: [
        authentication.hooks.authenticate(config.strategies)
      ],
      remove: [
        authentication.hooks.authenticate('jwt')
      ]
    }
  });
};

```

Set up a “Mixed Auth” Endpoint

Now we need to setup an endpoint to handle both unauthenticated and authenticated users. For this example, we'll use the `/users` service that was already created by the authentication generator. Let's suppose that our application requires that each `user` record will contain a `public` boolean property. Each record will look something like this:

```
{
  id: 1,
  email: 'my@email.com',
  password: "password",
  public: true
}
```

If a `user` record contains `public: true`, then **unauthenticated** users should be able to access it. Let's see how to use the `iff` and `else` conditional hooks from `feathers-hooks-common` to make this happen. Be sure to read the [iff hook API docs](#) and [else hook API docs](#) if you haven't, yet.

We're going to use the `iff` hook to authenticate users only if a token is in the request. The `feathers-authentication-jwt` plugin, which we used in `src/authentication.js`, includes a token extractor. If a request includes a token, it will automatically be available inside the `hook` object at `hook.params.token`.

src/services/users/users.hooks.js(This example only shows the `find` method's `before` hooks.)

```
'use strict';

const { authenticate } = require('feathers-authentication').hooks;
const commonHooks = require('feathers-hooks-common');

module.exports = {
  before: [
    {
      find: [
        // If a token was included, authenticate it with the `jwt` strategy.
        commonHooks.iff(
          hook => hook.params.token,
          authenticate('jwt')
        )
        // No token was found, so limit the query to include `public: true`
        .else( hook => Object.assign(hook.params.query, { public: true }) )
      ]
    }
  ];
};
```

Let's break down the above example. We setup the `find` method of the `/users` service with an `iff` conditional before hook:

```
iff(
  hook => hook.params.token,
  authenticate('jwt')
)
```

For this application, the above snippet is equivalent to the snippet, below.

```
hook => {
  if (hook.params.token) {
    return authenticate('jwt')
  } else {
    return Promise.resolve(hook)
  }
}
```

The `iff` hook is actually more capable than the simple demonstration, above. It can handle an `async` predicate expression. This would be equivalent to being able to pass a `promise` inside the `if` statement's parentheses. It also allows us to chain an `.else()` statement, which will run if the predicate evaluates to false.

```
.else( hook => Object.assign(hook.params.query, { public: true }) )
```

The above statement simply adds `public: true` to the query parameters. This limits the query to only find `user` records that have the `public` property set to `true`.

Wrapping Up

With the above code, we've successfully setup a `/users` service that responds differently to unauthenticated and authenticated users. We used the `hook.params.token` attribute to either authenticate a user or to limit the search query to only public users. If you become familiar with the [Common Hooks API](#), you'll be able to solve almost any authentication puzzle.

FeathersJS Auth Recipe: Set up Basic OAuth Login

The Auk release of FeathersJS includes a powerful new [authentication suite](#) built on top of [PassportJS](#). This now gives the Feathers community access to hundreds of authentication strategies from the Passport community. Since many of the Passport strategies are for OAuth, we've created two auth plugins, `feathers-authentication-oauth1` and `feathers-authentication-oauth2`. These new plugins use a Passport strategy to allow OAuth logins into your app.

Adding OAuth authentication to your app is a great way to quickly allow users to login. It allows the user to use an existing Internet account with another service to login to your app. Among lots of good reasons, it often eliminates the need for the email address verification dance. This is even more likely for very common OAuth providers, like GitHub, Google, and Facebook.

Simplified login is almost always a good idea, but for many developers implementing OAuth can be difficult. Let's take a look at how it works, in general. After that, we'll see how the new `feathers-authentication` server plugin makes it easy to get up and running.

How OAuth Works

There are a couple of different methods you can use to implement OAuth. Here are the basic steps of the flow that the `feathers-authentication-oauth1` and `feathers-authentication-oauth2` plugins use.

1. You register your application with the OAuth Provider. This includes giving the provider a callback URL (more on this later). The provider will give you an app identifier and an app secret. The secret is basically a special password for your app.
2. You direct the user's browser to the OAuth provider's site, providing the app identifier in the query string.
3. The content provider uses the app identifier to retrieve information about your app. That information is then presented to the user with a login form. The user can grant or deny access to your application.
4. Upon making a decision, the provider redirects the user's browser to the callback URL you setup in the first step. It includes a short-lived authorization code in the querystring.
5. Your server sends a request to the OAuth provider's server. It includes the authorization code and the secret. If the authorization code and secret are valid, the provider returns an OAuth access token to your server. Some user data can also be sent.
6. Your server can save the user information into the `/users` table. It can also use this access token to make requests to the provider's API. This same information can also be sent to the browser for use.
7. With Feathers, there is an additional step. After logging in, a JWT access token is stored in a cookie and sent to the browser. The client uses the JWT to authenticate with the server on subsequent requests.

Implementing OAuth with Feathers

The `Feathers-cli` allows you to easily setup a new application with OAuth. Here are the steps to generate an application:

1. `npm install -g feathers-cli`
or
`yarn global feathers-cli`
2. `mkdir feathers-demo-oauth; cd feathers-demo-oauth`
or a folder name you prefer.
3. `feathers generate app`
use the default prompts.

4. feathers generate authentication
 - o Select Facebook, GitHub, or Google when prompted for a provider.
This guide will show how to use GitHub.
 - o Select the defaults for the remaining prompts.

Setting up the OAuth Provider

To setup the provider, you use each provider's website. Here are links to common providers:

- [Facebook](#)
- [GitHub](#)
- [Google](#)

Once your app is setup, the OAuth provider will give you a `client ID` and `Client Secret`.

Configuring Your Application

Once you have your app's `Client ID` and `Client Secret`, it's time to setup the app to communicate with the provider. Open the `default.json` configuration file. The generator added a key to the config for the provider you selected. The below configuration example has a `github` config. Copy over and replace the placeholders with the `clientID` and `clientSecret`.

config/default.json

```
{
  "host": "localhost",
  "port": 3030,
  "public": "./public/",
  "paginate": {
    "default": 10,
    "max": 50
  },
  "authentication": {
    "secret": "cc71e4f97a80c878491197399aabf74e9c0b115c9f8071e75b306c99c891a54b7171852f8c5508e1fe4dcfaedb60317
8b0935261928592e487e628f2f669f3a752f2beb3661b29d521b36c8a39e1be6823c0362df5ef1e212d7f2daae789df1065293b98ec9b43
309ffe24dba3a2ec2362c5ce5c9155c6438ec380bc7c56d6a169988c0f6754077c5129e8a0ee5fd85b2182d87f84312387e1bbebe49ad
1bf2dcf783e7d8cbee40272b141358b8e23150eee5ea8fc04b2a0f3d824e7fa9d46c025c619c3281af91b7a19fd760bcceda379b735c85
024b25a9c91749935b2f29d5b69b2c1ff29368b4aa9cf426d9960302e5e7b903d53e18ccbe2325cf3b6",
    "strategies": [
      "jwt"
    ],
    "path": "/authentication",
    "service": "users",
    "jwt": {
      "header": {
        "type": "access"
      },
      "audience": "https://yourdomain.com",
      "subject": "anonymous",
      "issuer": "feathers",
      "algorithm": "HS256",
      "expiresIn": "1d"
    },
    "github": {
      "clientID": "your github client id", // Replace this with your app's Client ID
      "clientSecret": "your github client secret", // Replace this with your app's Client Secret
      "successRedirect": "/"
    },
    "cookie": {
      "enabled": true,
      "name": "feathers-authentication"
    }
  }
}
```

```

        "name": "feathers-jwt",
        "httpOnly": false,
        "secure": false
    },
},
"nedb": "../data"
}

```

Test Login with OAuth

Your app is ready for OAuth logins. We've made it that simple! Let's try it out. Open the file `public/index.html` and scroll to the bottom. Add the following code just under the `h2`:

```

<p class="center-text"><br/>
<a href="/auth/github">Login With GitHub</a>
</p>

```

Now add the following code to the same page. The first script tag loads Feathers Client from a CDN. The second script loads Socket.io. The third script creates a Feathers Client instance and attempts to authenticate with the JWT strategy upon page load. The authentication client plugin has been configured with a `cookie` value of `feathers-jwt`.

Note: This code loads the `feathers-client` package from a CDN. This is **not** the recommended usage for most apps, but is good for demonstration purposes. We recommend using a bundler as described in the [Feathers Client API docs](#).

```

<script src="//unpkg.com/feathers-client@2.0.0/dist/feathers.js"></script>
<script src="//unpkg.com/socket.io-client@1.7.3/dist/socket.io.js"></script>
<script>
// Socket.io is exposed as the `io` global.
var socket = io('http://localhost:3030', {transports: ['websocket']});
// feathers-client is exposed as the `feathers` global.
var feathersClient = feathers()
.configure(feathers.hooks())
.configure(feathers.socketio(socket))
.configure(feathers.authentication({
  cookie: 'feathers-jwt'
}));

feathersClient.authenticate()
.then(response => {
  console.info('Feathers Client has Authenticated with the JWT access token!');
  console.log(response);
})
.catch(error => {
  console.info('We have not logged in with OAuth, yet. This means there\'s no cookie storing the accessToken. As a result, feathersClient.authenticate() failed.');
  console.log(error);
});
</script>

```

Now, run the server, open `http://localhost:3030`. Before you click the "Login with GitHub" link, open the console. If you refresh you'll see the message in the catch block. Since we haven't logged in, yet, we don't have a stored JWT access token. Now, click the `Login with GitHub` button. Assuming you haven't logged in to Github with this application, before, you'll see a GitHub login page. Once you login to GitHub, you'll be redirected back to `http://localhost:3030`. Now, if you look at your console, you should see a success message.

What just happened? When you clicked on that link, it opened the `/auth/github` link, which is just a shortcut for redirecting to GitHub with your `Client ID`. The entire OAuth process that we described earlier took place. The browser received a `feathers-jwt` cookie from the server. Finally the script that we added in the last step used the

`feathers-authentication-client` to authenticate using the JWT returned from the server. There were a lot of steps that happened in a very short time. The best news is that you're authenticated with OAuth.

Wrapping Up

You've now seen how OAuth login is greatly simplified with the new Feathers Authentication plugins. Having plugins built on top of PassportJS allows for a lot of flexibility. You can now build nearly any authentication experience imaginable. In the final part of this guide, you were able to authenticate the Feathers client. Hopefully this will get you started integrating OAuth into your application.

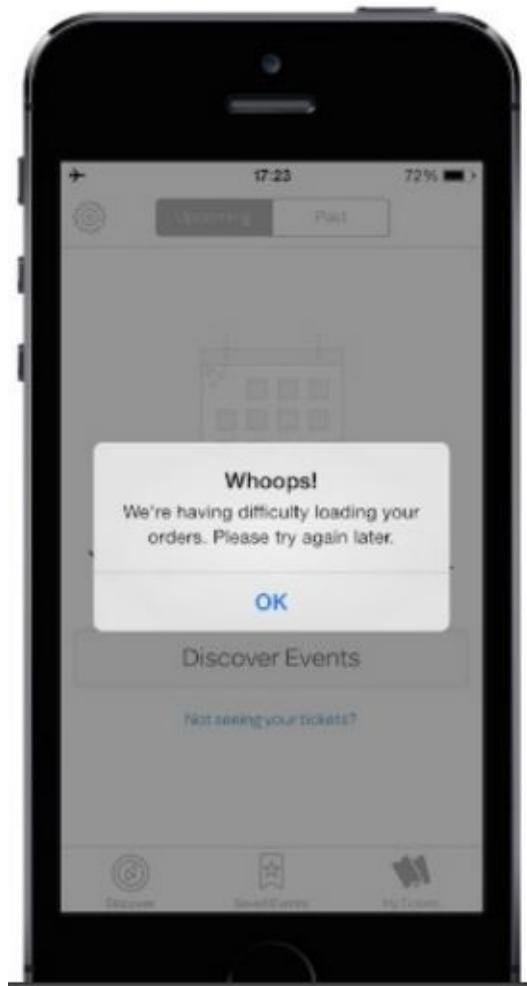
Offline-first

The Landscape

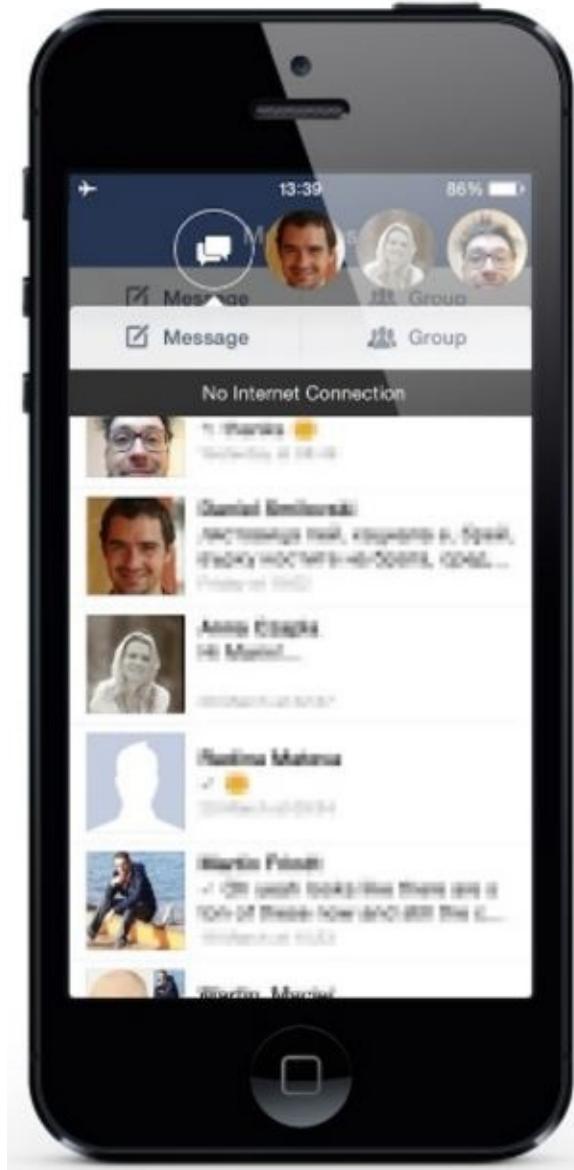
Customers are increasingly purchasing goods and services using smartphones, either through native or web applications. And, they want to make these purchases wherever they may be, whether it's inside trains, in the countryside or roaming in foreign countries.

However, mobile networks are still known to be unreliable. Even in highly covered areas and with modern networks, you still encounter high latency and network failure rates. But it's not just the network at fault here: the customer experience for failure rates also depends on how a given application deals with being offline.

In one extreme end of the quality spectrum, you have an application that, when a network error occurs, the application presents the user with a crude error message and no clue of how to recover.



On the opposite end of the spectrum are applications that automatically deal with network failures — this is the good end of the spectrum.



(*)

What is Offline-first?

Offline-first is a way of building applications where having network connectivity is an enhancement, not a necessity. Instead of building applications with the always-on desktop mindset, when you build applications in which the default mode is offline, you're prone to deliver a better overall customer experience.

Offline-first techniques and technologies exist to prepare an application to deliver a good experience to customers while it's offline.

(*)

Client has Local Data

The only way that an application can access data while it's offline is, of course, to store that data locally.

Your application prefers reading from the local source, while that local data is updated from remote sources.

The First Generation of Offline-first (*)

Some database technologies exist that work on the client side. SQLite is the reference of embeddable databases and it's often used on native applications. In the web realm, PouchDB offers a document store on top of the storage the browser offers. PouchDB also has some nice features — it can sync with a back-end CouchDB server, Cloudant, a PouchDB Node.js server or any other database that implements the CouchDB replication protocol.

In this architecture, each client has its own dedicated database, which is then replicated to a dedicated database on the back-end. Each database may then contain the customer documents. (A document is a JSON object that may contain any arbitrary data).

One database per customer may sound strange to people who are used to relational databases, but it's a usual pattern when using CouchDB and variants. It's also a way to clearly and naturally separate and enforce which data a user has access to.

When a change happens on the client or on the server, the sync protocol kicks in and tries to replicate that change to the other side. If no network connection is possible at that time, the client will try to reconnect. Once a connection is possible, both databases will be able to talk to each other and synchronize.

In this architecture, both the client and the server can make changes to the data concurrently. If a conflict arises for a given document, the replication protocol makes sure that both databases converge to the same version of that document. When a conflict happens, Pouch and CouchDB keep all the conflict data around. If the programmer so wishes, they can solve that conflict with any strategy they deem correct and that minimizes data loss.

(*)

Issues

One database per customer is logically reasonable for many mobile applications as customers don't interact in such applications. However a successful application may end up having thousands of databases, one for each of its thousands of customers. You can use tools, such as those provided by IBM, which maintain only one database but makes it look like many individual CouchDB databases.

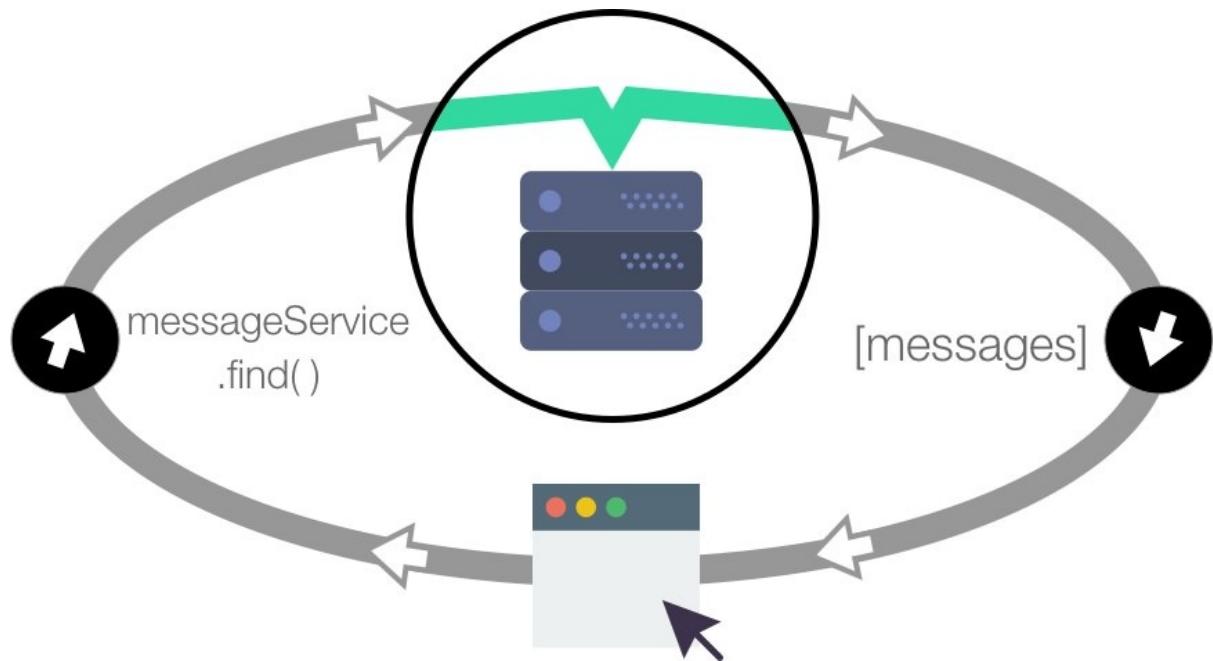
A second issue is that the remote and local databases synchronize by replication. The only way to make replication realtime is to start a synchronization cycle every time any data changes on either the remote or the local database.

This is overhead to consider even when you have one database per client. When the same data is shared by clients, providing realtime updates to all of them is problematic, as a replication cycle is needed for each client.

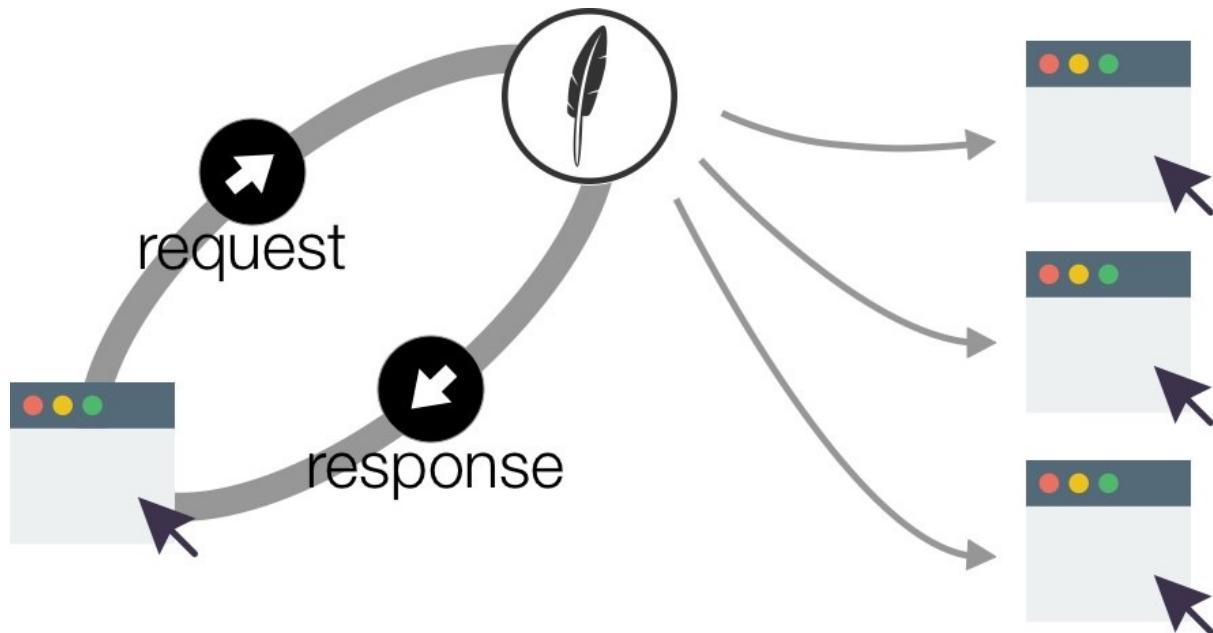
Feathers Offline-first

Feathers has unique foundational features which are useful for implementing offline-first.

First, local applications can easily mutate remote databases. This means we can add a hook to a local database so that, whenever it is locally mutated, the hook can optimistically mutate the remote database.



Second, local applications know when a remote database is mutated as that mutation emits a Feathers realtime event on the client.



Feathers offline-first therefore reacts in realtime when connected, employing a replication strategy only when disconnected. This makes applications using it less sluggish.

Importantly, Feathers offline-first allows a client database to reflect only a portion of the remote database. So information for all customers can reside in one database, with each client seeing only those items its allowed to.

No universal solution

There is no one universal solution to implementing offline-first. The end.

For each application, you have to:

- assess the problem

- determine the correct solution
- implement the right code

In this order please!

Many mobile applications only need a read-only local database which is infrequently refreshed. Why would you implement a complex replication strategy in such cases?

Feathers offline-first provides several strategies for implementing offline-first. Determine your application's needs and then choose the simplest strategy which satisfies them.

Sources:

- (*) [Pedro Teixeira](#)

Strategies

Comparison of Strategies

Feathers offline-first provides several increasingly sophisticated strategies. It's generally straightforward to change your application to use a more sophisticated one (except for snapshot).

ProTip: The snapshot and realtime (with optimistic mutation while connected) strategies are available at this time.

The features for each strategy are shown below.

Feature.....	snap shot	real time	optimistic mutation	own-data own-net	sync-data sync-net	time-travel
Replicate partial table						
- using query syntax	Y	Y	Y	Y	Y	
- using JS functions	-	Y	Y	Y	Y	
Snapshot data on connect	Y	Y	Y	Y	Y	
Is a <code>uuid(1)</code> field required?	-	-	Y	Y	Y	
Remote changes mutate client	-	Y	Y	Y	Y	
- minimal service events	-	Y	Y	Y	Y	
Client can mutate remote data						
- with remote service calls	-	Y	Y	Y	Y	
- optimistic client mutation	-		Y	Y	Y	
Keep queue while disconnected						
- Keep every call	-	-	-	own-data	sync-data	
- Only record net change	-	-	-	own-net	sync-net	
Process queue on reconnection	-	-	-	Y	Y	
- Conflict resolution handling	-	-	-	-	Y	
Snapshot data on reconnect	-	Y	Y	Y	Y	
Repository	(2)	(3)	(4)	tba	tba	tba

ProTip: Note that the realtime strategy supports optimistic mutation only while connected.

ProTip: It is also your responsibility to inform the replicator when a connection is (re)established or lost using `replicator.connect()` and `replicator.disconnect()`. A repo handling this for both browser and react native is planned but not yet available.

- (1) [Universally unique identifier \(uuid\)](#)
- (2) [feathers-offline-snapshot](#)
- (3) [feathers-offline-realtime](#)
- (4) [feathers-offline-realtime with /optimistic-mutator](#)

Snapshot

What is the Snapshot strategy?

Snapshot distributes data exactly as it appears at a specific moment in time and does not monitor for updates to the data. When synchronization occurs, the entire snapshot is generated and sent to client service.

Snapshot can be used by itself, but the snapshot process is also commonly used to provide the initial set of data for all the other strategies.

Using Snapshot by itself is most appropriate when one or more of the following is true:

- Data changes infrequently.
- It is acceptable to have copies of data that are out of date with respect to the remote service for a period of time.
- Replicating small or medium volumes of data.
- A large volume of changes occurs over a short period of time.
- Keep the current values after having lost connection for some time.

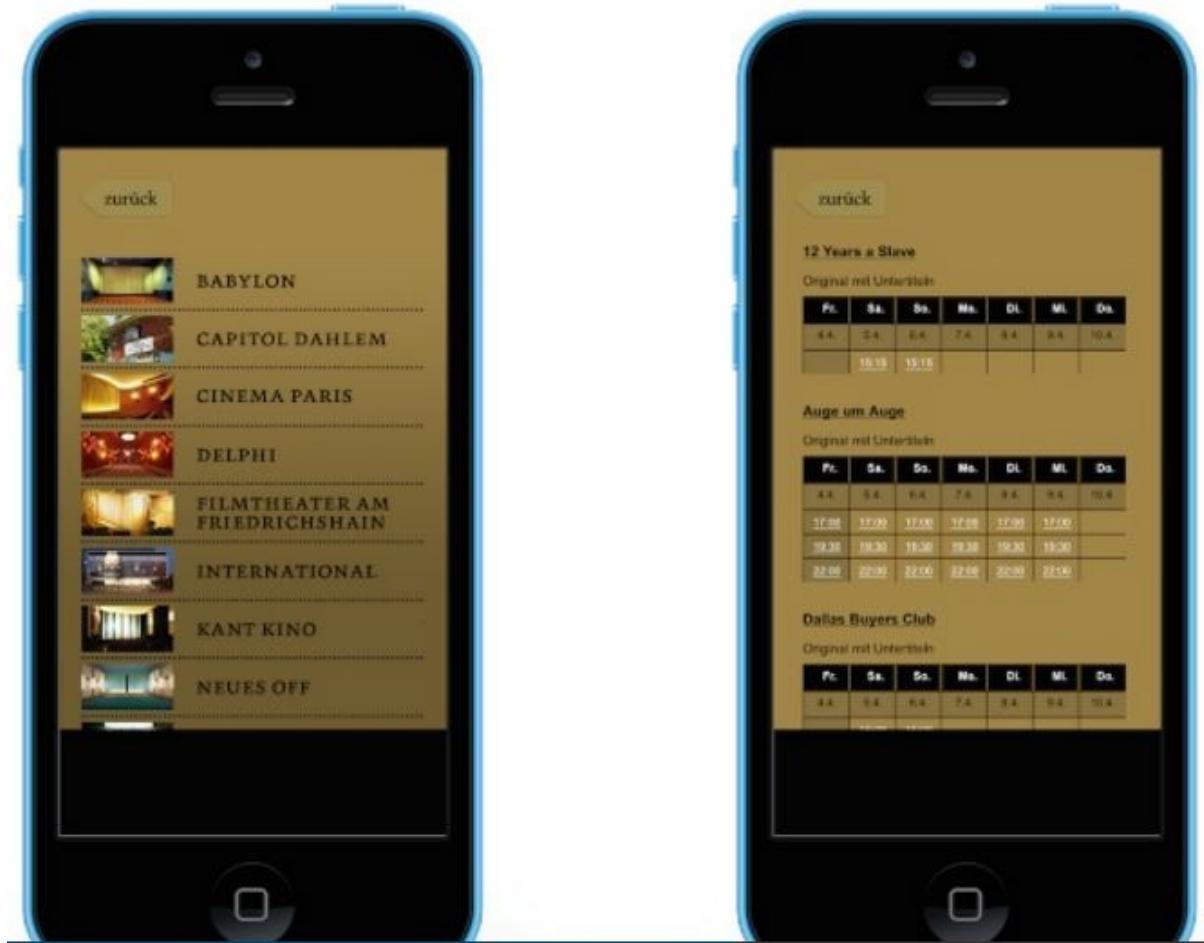
Snapshot replication is most appropriate when data changes are substantial but infrequent. For example, if a sales organization maintains a product price list and the prices are all updated at the same time once or twice each year, replicating the entire snapshot of data after it has changed is recommended. Given certain types of data, more frequent snapshots may also be appropriate. For example, if a relatively small table is updated on a remote service during the day, but some latency is acceptable, changes can be delivered nightly as a snapshot.

Snapshot has a lower continuous overhead on both the client and the remote than the other strategies, because incremental changes are not tracked. However, if the dataset set is very large, it will require substantial resources to generate and apply the snapshot. Consider the size of the entire data set and the frequency of changes to the data when evaluating whether to utilize snapshot replication.

(*)

Snapshot Case Study

Let's consider a mobile application for movie cinemas, which lists show times.



Let's assess the data problem.

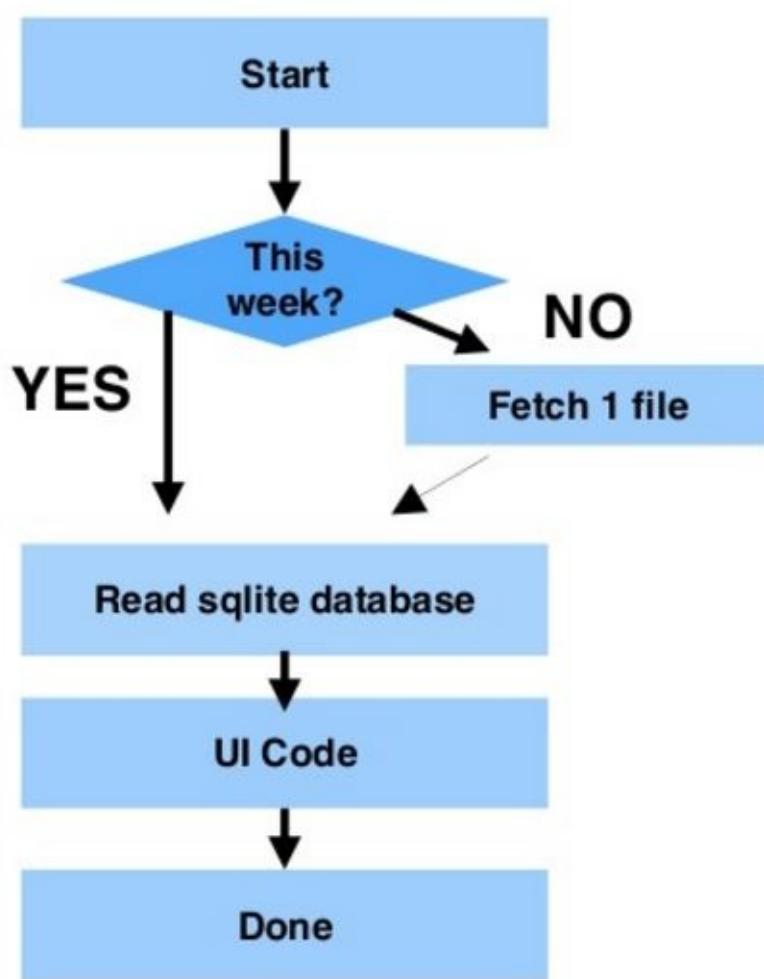
- Cinemas change once a year.
- The schedule changes every Thursday.
- The static information rarely changes.
- We need to support ticket reservations.
- We don't need past data.

Let's look at the data. We need:

- 12 cinema photos + their names + coordinates
- 25 film posters + film names
- 30 movie times for each cinemas

This is about 7k of files and 2M of photos, which takes less space than 1 Facebook photo.

The problem definition contains the solution:



The snapshot strategy is the simplest one to satisfy the needs.

How about ticket reservations? You got to do some things online! However, you can still give a telephone number.

(**)

Sources:

- (*) Microsoft
- (**) MarinTodorov

Realtime

What is the Realtime strategy?

Realtime is also commonly called transactional replication.

Realtime typically starts with a snapshot of the remote database data. As soon as the initial snapshot is taken, subsequent data changes made at the remote are delivered to the client as they occur (in near real time). The data changes are applied at the client in the same order as they occurred at the remote.

Realtime is appropriate in each of the following cases:

- You want incremental changes to be propagated to clients as they occur.
- The application requires low latency between the time changes are made at the remote and the changes arrive at the client.
- The application requires access to intermediate data states. For example, if a row changes five times, realtime allows an application to respond to each change (such as running hooks), not simply to the net data change to the row.
- The remote has a very high volume of create, update, patch, and remove activity.

Realtime Case Study

Let's consider an application which shows historical stock prices.



The realtime strategy would snapshot the initial historical data. It would then update the local data with every addition or other mutation made on the remote.

ProTip: You should check that `replicator.connected === true` before doing a mutation either by calling the remote service directly or using the optimistic mutator. You could display an app-wide status message on the UI while disconnected, for example "There is no connection to the server. Only inquiries are allowed at the moment."

Sources:

- (*) Microsoft
- (**) MarinTodorov

Realtime with Optimistic Mutation

Realtime replication only replicates when the client has a connection to the server. It also requires the client be connected to the server with WebSockets.

Using the remote service

You can start a realtime replication and mutate the records on the remote service.

```
import Realtime from 'feathers-offline-realtime';
const messages = app.service('/messages');

const messagesRealtime = new Realtime(messages, { ... });

messagesRealtime.connect()
  .then(() => messages.create(...))
  .then(data => ...);
```

The remote service mutates the data; its service filter sends the event; the realtime replicator receives that service event and updates the client replica.

ProTip: This is a straight forward, effective and simple approach.

In this scenario, the replicator would emit the following [event](#):

action	eventName	record	records	source	description
mutated	created	yes	yes	0	remote service event

Mutation delays

There is a delay between the client running `messages.create(...)` and the client replica containing the new record. The service call must be transmitted to the server via a WebSocket. The remote service must make the database call. The database must schedule and perform it. The service filters on the server must transmits the event to the client. The replicator process the service event, and then finally updates the client replica.

Much of the time this delay is acceptable. However sometimes it may not be, particularly on mobile devices with their relatively slow connections.

The realtime replicator's **optimistic mutation** may be used to produce a **snappier** response at the client.

Optimistic mutation

Using optimistic mutation is similar to using the remote service directly.

```
import Realtime from 'feathers-offline-realtime';
import optimisticMutator from 'feathers-offline-realtime/optimistic-mutator';
const messages = app.service('/messages');

const messagesRealtime = new Realtime(messages, { uuid: true });

app.use('clientMessages', optimisticMutator({ replicator: messagesRealtime }));
const clientMessages = app.service('clientMessages');
```

```
messagesRealtime.connect()
  .then(() => clientMessages.create({ ... }))
  .then(data => ...);
```

However what ensues is rather different. The optimistic-mutator service immediately updates the client replica to what it **optimistically** expects the final result will be, and **the user can see the change right away**. The replicator then emits an event.

Next, the same processing occurs as for a remote service call: the call to the server, the database processing, the filter, the service event on the client. Finally the replicator replaces the optimistic copy of the record with the one provided by the server. The replicator emits another event once this happens.

But what happens if the remote service rejects the mutation with an error? The replicator has kept a copy of the record from before the mutation and, once it detects the error response, it replaces the optimistic copy of the record with the prior version. The replicator emits a different event when this happens.

In a successful optimistic mutation, the replicator emits these [events](#):

action	eventName	record	records	source	description
mutated	created	yes	yes	1	optimistic update
mutated	created	yes	yes	0	remote service event

When the remote service returns an error, the replicator emits:

action	eventName	record	records	source	description
mutated	created	yes	yes	1	optimistic update
remove	removed	yes	yes	2	remote service error

Using data in the client replica

The optimistic mutator adapter has `find` and `get` methods which support the same feature set as `feathers-memory`. This makes it a great way to retrieve data from the client replica.

```
clientServices.find({ query: { username: 'john', $sort: { ... } } })
  .then(data => ...);
```

You can also access the client replica [directly](#).

Hooks

The remote service may run before and after hooks, and these may affect the data returned. The optimistic value of the record in the client replica may end up being replaced with something different.

ProTip: This may not make a difference in many use cases.

If this is problematic, hooks can be defined **on the client** for the optimistic-mutation service to reflect what the remote service does.

If this is still unsatisfactory, you can always make direct remote service calls and live with the latency.

uuid

Optimistic mutation requires that the records contain a `uuid` property. (It's the only way the replicator can match an optimistic create to the created service event.)

ProTip: The `id` value for optimistic mutation service calls must be the value of the `uuid` property in the data.

The optimistic mutation service's `create` method will automatically add a `uuid` property if it does not find one. You can configure whether an industry standard 32-char uuid is used, or if a shorter uuid-like value is used.

ProTip: There is virtually [no chance of collision](#) with the shorter value unless you work at high scale.

You can configure use of the 32-char uuid, instead of the default shorter value, with:

```
const messagesRealtime = new Realtime(messages, { ... });
messagesRealtime.useShortUuid(false);
```

ProTip: The replicator can provide you with uuid's for other purposes with `messagesRealtime.getUuid()`.

Own-data and Own-net

What are the own-data and own-net strategies?

The core data for many mobile applications is unique to the user using the application. Since no one other than the user can change that user's data, the client can safely mutate the remote data without concern that anyone is doing the same at that same time.

While the client is disconnected, both strategies queue mutation events for later processing when reconnected. The difference between them is what they queue.

Own-data queues every mutation event on the client service, and it will later process each mutation in order on the remote service. The remote service can react to every mutation. It may, for example, run hooks which send emails on certain mutations.

Own-net queues the net change for each record. If a record is patched 5 times, own-net queues the record contents after the last of the changes. If a record is created, patched and finally removed, the remote service will not see the mutations at all.

Own-net uses less storage on the client, and it reduces the load on the remote service upon reconnection.

Once the queue is processed, a snapshot refreshes the client's replica, bringing it up to date.

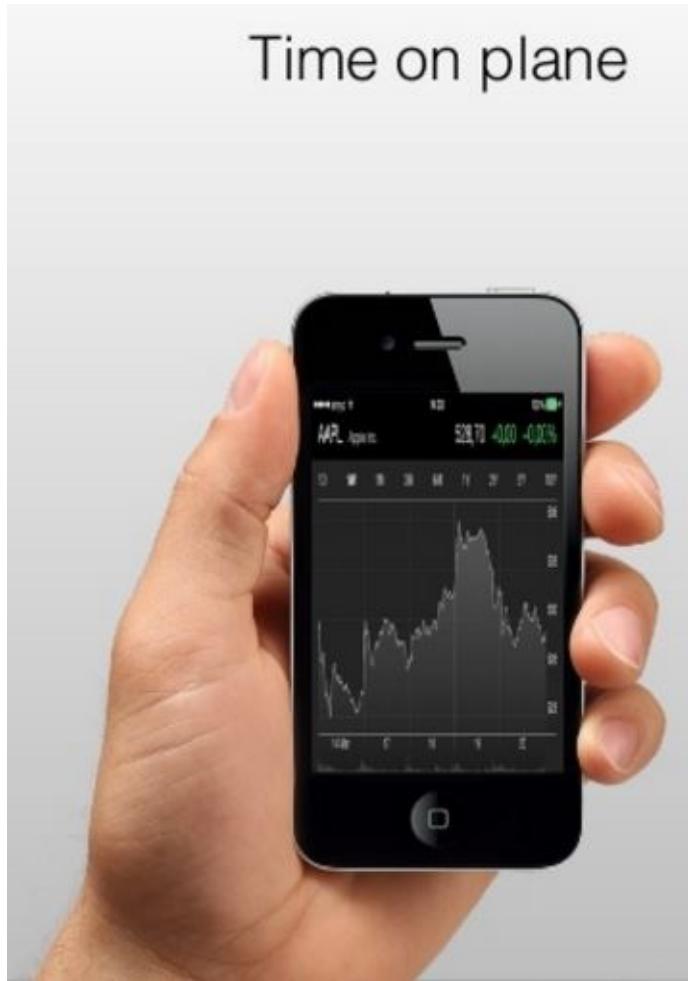
Own-data Case Study

The realtime case study involved displaying historical stock prices. Let's now allow the user to buy shares for his own portfolio.

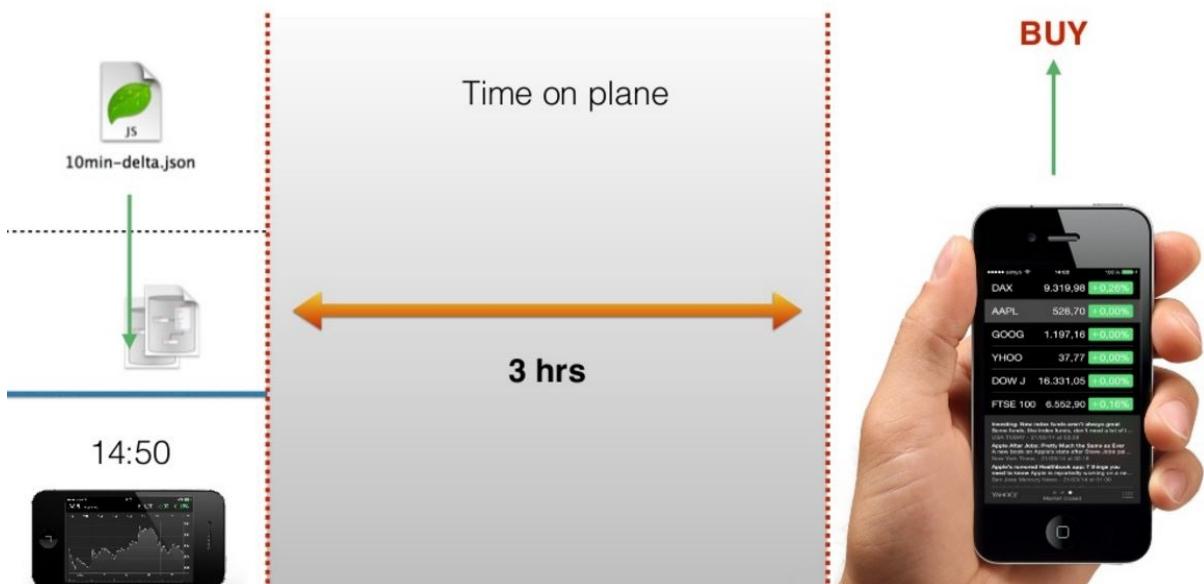


The user will be disconnected if he takes a plane trip, and he will no longer receive stock price updates.

Time on plane



However he can still make stock purchases. They will be sent to the server when he reconnects after landing.



Sources:

- (***) [MarinTodorov](#)

Sync-data and Sync-net

What are the sync-data and sync-net strategies?

More complex applications may allow multiple users to mutate the same data on the remote service. sync-data and sync-net are similar to own-data and own-net, however they detect mutation conflicts and help the server resolve them.

Each queue entry contains the contents of the record before the mutation. So each queue-entry for sync-data contains what the record contained on the client before each mutation. Each entry for sync-net contains what the record contained before the series of mutations.

The remote service, when processing the queued entries, compares what the client service record contained before the mutation to what the remote service's record currently contains.

If the two are the same, then the client changed a record identical to what the remote currently has. There is no conflict and the mutation may be performed on the remote.

Let's say the user of one client device went on a plane trip and was offline for some time. During that time the user changed the addresses of several branch offices, in particular the NYC one from `100 Lexington` to `123 Fifth Avenue`. In the meantime another user changed the address to `123 5-th Ave.`.

The queued mutations will be processed once the user reconnects after his flight. The only conflict to occur will be for the NYC office. The mutation from `100 Lexington` to `123 Fifth Avenue` conflicts with the record now being `123 5-th Ave.` on the remote service.

Client app					Server		
Client	Branch	Address was	Address changed to	Connection	Address before	Conflict detected?	Address after
1	NYC	100 Lexington	123 Fifth Avenue	disconnected	100 Lexington	no	100 Lexington
2	NYC	100 Lexington	123 5-th Ave.	connected	100 Lexington	no	123 5-th Ave.
1	NYC			reconnected	123 5-th Ave.	yes	?

The remote service's `conflict resolver function` will be called with:

- the remote service's current record.
- the client service's method call, e.g. 'create', 'update', ...
- the client service's query object.
- the client service's before record.
- the record the client service wants to mutate to.

Input to conflict resolver:

Remote service's		
Current record	NYC	123 5-th Ave.
Client service's		
Method call	patch	
Query object	{}	
Before record	NYC	100 Lexington
Resulting record	NYC	123 Fifth Avenue

The resolver may indicate:

- the current remote service's record is to be retained.

- the remote service's record should be mutated to the client service's record.
- what the contents of the record should now be.

Conflict resolver's choices:

Use remote service's current record	NYC	123 5-th Ave.
Use client service's proposed record	NYC	123 Fifth Avenue
Use business rules to determine result	NYC	123 Fifth Ave.

This allows the remote service great control over conflicts. However, in your particular use case, it may be appropriate to adapt a last-mutation-wins strategy and also use the client service's mutation.

Other use cases may require analyzing the contents of the two records and applying business rules.

Sometimes own-data is sufficient

Let's say a portion of your [Neighborhood Watch](#) app allows residents to report the location of a suspect in realtime, while the neighborhood waits for law enforcement to arrive. Your app may allow residents to update the suspect's location by using the mobile's geolocation.



You may decide, after assessing the application needs and legal requirements, to maintain only one record which contains only the suspect's latest location. All the residents will be updating this one record.

In this use case, although many clients are updating the same record, you might not care about detecting mutation conflicts among the various clients. The app just cares about the latest sighting.

Client app				Server		
User	Suspect's location on their client	Time they saw the suspect	Where they record seeing the suspect	Suspect's location on server before	Conflict detected?	Suspect's location on server after
John	37, -122	17:00	38, -123		no	38, -123
Jane	37, -122	17:02	39, -124	38, -123	yes	39, -124
Jack	39, -124	17:05	40, -125	39, -124	yes	40, -125
Jenny	39, -124	17:07	41, -126	40, 125	yes	41, -126

You have two choices.

- You could use the sync-data strategy with the conflict resolver always choosing the client services's latest record, or
- You could use the own-data strategy and let the client mutations update over one another based solely on when the clients do the mutation.

Whenever you decide to always use the client mutation in the conflict resolver, consider switching from the sync-strategy to a own- one, as in such cases the results are the same.

Keep things simple

You don't always have to choose the most sophisticated strategy. Its harder to reason about designs which are more sophisticated. There is a greater possibility of introducing errors. It requires a software maintainer understand more things before working with the code. Finally, in this case, it increases the processing load on both the server and the client.

Often the best course is to choose the simplest strategy which addresses the needs.

Configure Snapshot

Installation

```
npm install feathers-offline-snapshot --save
```

Documentation

```
import snapshot from 'feathers-offline-snapshot';
snapshot(service, query).then(records => ...);
```

- `service` (*required*) - The service to read.
- `query` (*optional*, default: `{}`) - The [Feathers query object](#) selecting the records to read. Some of the props it may include are:
 - `$limit` (*optional*, default: 200) - Records to read at a time. The service's configuration may limit the actual number read.
 - `$skip` (*optional*, default: 0) will initially skip this number of records.
 - `$sort` (*optional*, default: `{}`) will sort the records. You can sort on multiple props, for example `{ field1: 1, field2: -1 }`.

Example

```
const snapshot = require('feathers-offline-snapshot');

const app = ... // Configure Feathers, including the `/messages` service.
const username = ... // The username authenticated on this client
const messages = app.service('/messages');

snapshot(messages, { username, $sort: { channel: 1 } })
  .then(records => {
    console.log(records);
  });

```

Configure Realtime, with optimistic mutation

You can keep on the client a near realtime replica of (some of) the records in a service configured on the server.

You can optimistically create, modify and remove records in the client replica using standard Feathers service calls. These mutations are also asynchronously performed on the server, and those delayed results may themselves mutate the client replica. Any errors on the server will revert the data in the client replica.

These features may make your client more performant, so it appears "snappier."

You can replicate just a subset of the records in the service by providing an optional "publication" function which, given a record, determines if the record belongs in the publication. The publication function may be as complicated as you need though it must be synchronous.

You or some other party may update a record so that it no longer belongs to the publication, or so that it newly belongs. The replicator handles these situations.

Many apps have unique data for every user. With publications, you can keep the records for all users in one table, using the publication feature to replicate to the client only those records belonging to the client's user.

A [snapshot replication](#) is used to initially obtain the records.

The realtime replicator can notify you of data mutations by emitting an event and/or calling a subscription function for every notification. You can in addition periodically poll the replicator to obtain the current realtime records.

ProTip: By default, the client will receive every service event. You may however use `feathers-offline-publication`, as mentioned below for `new Realtime(service, options)`, to reduce the number of service events received by the client to a minimum. This may noticeably improve performance, especially on mobile devices, as the client will consume less bandwidth.

ProTip: You can also [filter these events](#) manually.

You can control the order of the realtime records in the client replica by providing a sorting function compatible with `array.sort(...)`. Two sorting functions are included in this repo for your convenience:

- `Realtime.sort(fieldName)` sorts on the `fieldName` in ascending order.
- `Realtime.multiSort({ fieldName1: 1, fieldName2: -1 })` sorts on multiple fields in either ascending or descending order.

You can dynamically change the sort order as your needs change. This can be useful for your UI.

Snapshot performance

By default, the publication function will be run against every record in the service during a snapshot. This may lead to inefficiencies should, for example, a service contain records for 1,000 users and you want to replicate just the records for just one of them.

To avoid such situations, you may provide a [Feathers query object](#), suitable for use in a `.find({ query })` call, to reduce the number of records read initially. The publication function will still be run against the returned records.

ProTip: A publication function is required whenever you provide the query object, and the publication must be at least as restrictive as the query.

ProTip: You may find it convenient to use publication functions with the same query object as their input. For example `publication: require('sift').sift({ username: 'John' })`.

Installation

```
npm install feathers-offline-realtime --save
```

Documentation

Realtime read-only replication:

```
import Realtime from 'feathers-offline-realtime';
const messages = app.service('/messages');

const messagesRealtime = new Realtime(messages, options);

messagesRealtime.connect()
  .then(() => {
    console.log(messagesRealtime.connected);
    messagesRealtime.changeSort(Realtime.multiSort(...));
  });
}
```

Realtime replication with optimistic mutation:

```
import Realtime from 'feathers-offline-realtime';
import optimisticMutator from 'feathers-offline-realtime/optimistic-mutator';
const messages = app.service('/messages');

const messagesRealtime = new Realtime(messages, Object.assign({}, options, { uuid: true }));

const app = feathers() ...
app.use('clientMessages', optimisticMutator({ replicator: messagesRealtime }));
const clientMessages = app.service('clientMessages');

messagesRealtime.connect()
  .then(() => clientMessages.create({ ... }))
  .then(record => {
    console.log(messagesRealtime.connected, record);
    messagesRealtime.changeSort(Realtime.multiSort(...));
  });
}
```

Options: new Realtime(service, options) - Create a realtime replicator.

- `service (required)` - The service to read.
- `options (optional)` - The configuration object.
 - `publication (optional but required if query is specified)` - Function with signature `record => boolean` - Function to determine if a record belongs to the publication.
 - `query (optional)` - The [Feathers query object](#) to reduce the number of records read during the snapshot. The props \$limit, \$skip, \$sort and \$select are not allowed.
 - `sort (required Function with signature (a, b) => 1 || -1 || 0)` - A function compatible with `array.sort(...)`.
 - `subscriber (optional Function with signature (records, { action, eventName, source, record }) => ...)` - Function to call on mutation events. See example below.
 - `uuid (optional boolean)` - The records contain a `uuid` field and it should be used as the key rather than `id` or `_id`. `uuid: true` is required when optimistic mutation is being used.

ProTip: You may want to use some of the common publications available in [feathers-offline-publication](#).

ProTip: You can use `clientPublications.addPublication(clientApp, serviceName, options)` from `feathers-offline-publication`. That will not only return a suitable function for `production`, but it will also minimize the number of service events received by the client. This may noticeably improve performance, especially on mobile devices, as the client will consume less bandwidth.

Options: connect() - Create a new snapshot and start listening to events.

Options: disconnect() - Stop listening to events. The current realtime records remain.

Options: connected - Is the replicator listening to Feathers service events?

Options: changeSort(sorter) - Change the sort used for the records.

- `sorter` (*required*) - Same as `options.sort`.

Options: Realtime.sort(name) - Suitable for use with `array.sort(...)`. Sort on a field in ascending order.

- `name` (*required*) - The name of the field to sort on.

Options: Realtime.multiSort(sortDefn) - Suitable for use with `array.sort(...)`. Sort on multiple fields, in ascending or descending order.

- `sortDfn` (*required*) - Has the format `{ fieldName: order, ... }`.
 - `fieldName` (**required*) - The name of the field to sort on.
 - `order` (*required*) - Use 1 for ascending order, -1 for descending.

Options: app.use(path, optimisticMutator({ replicator })) - Configure a service to optimistically mutate the client replica while asynchronously mutating on the server.

- `replicator` (*required*) - The handle returned by the replicator.
- `paginate` (*optional*) - A [pagination object](#) containing a default and max page size.

ProTip: The `id` value for these service calls must be the value of the `uuid` property in the data.

The `create` method adds a `uuid` property to the data if none is provided. By default this will be a short, but variable-length, random string. There is virtually [no chance of collision](#) unless you work at high scale.

You can change the default to use the standard 32-char uuid values by running

```
const messagesRealtime = new Realtime(messages, { ... });
messagesRealtime.useShortUuid(false);
```

ProTip: Two events are emitted for each optimistic mutation of the client replica. The first occurs when the client replica is mutated. It is identified by `source = 1` (see Event information below). A successful server mutation produces another event having `source = 0`. A failed server mutation reverts the record in the client replica back to its original value. That produces an event having `source = 2`.

Example using event emitters

```
const Realtime = require('feathers-offline-realtime');

const app = ... // Configure Feathers, including the `/messages` service.
const username = ... // The username authenticated on this client
const messages = app.service('/messages');

const messagesRealtime = new Realtime(messages, {
  query: { username },
  publication: record => record.username === username && record.inappropriate === false,
  sort: Realtime.multiSort({ channel: 1, topic: 1 }),
```

```

});
```

```

messagesRealtime.on('events', (records, { action, eventName, record }) => {
  console.log('last mutation:', action, eventName, record);
  console.log('realtime records:', records);
  console.log('event listeners active:', messagesRealtime.connected);
});
```

```

messagesRealtime.connect()
  .then(() => ...);
```

Example using a subscriber

```

const messagesRealtime = new Realtime(messages, {
  query: { username },
  publication: record => record.username === username && record.inappropriate === false,
  sort: Realtime.multiSort({ channel: 1, topic: 1 }),
  subscriber
});
```

```

messagesRealtime.connect()
  .then(() => ...);
```

```

function subscriber(records, ({ action, eventName, record }) => {
  console.log('last mutation:', action, eventName, record);
  console.log('realtime records:', records);
  console.log('event listeners active:', messagesRealtime.connected);
})
```

Example using periodic inspection

```

const messagesRealtime = new Realtime(messages, {
  query: { username },
  publication: record => record.username === username && record.inappropriate === false,
  sort: Realtime.multiSort({ channel: 1, topic: 1 }),
});
```

```

setTimeout(() => {
  const { records, last: { action, eventName, record } } = messagesRealtime.store;
  console.log('last mutation:', action, eventName, record);
  console.log('realtime records:', records);
  console.log('event listeners active:', messagesRealtime.connected);
}, 5 * 60 * 1000);
```

Example using a publication with a query object

```

const Realtime = require('feathers-offline-realtime');
const sift = require('sift');

const app = ... // Configure Feathers, including the `/messages` service.
const username = ... // The username authenticated on this client
const messages = app.service('/messages');
const query = { username };

const messagesRealtime = new Realtime(messages, {
  query,
  publication: sift(query),
  sort: Realtime.multiSort({ channel: 1, topic: 1 }),
```

```

});

messagesRealtime.on('events', (records, { action, eventName, record }) => {
  console.log('last mutation:', action, eventName, record);
  console.log('realtime records:', records);
  console.log('event listeners active:', messagesRealtime.connected);
});

messagesRealtime.connect()
.then(() => ...);

```

Event information

All handlers receive the following information:

- `action` - The latest replication action.
- `eventName` - The Feathers realtime service event.
- `source` - Cause of mutation:
 - 0 = service event.
 - 1 = optimistic mutation.
 - 2 = revert to original record when an optimistic mutation results in an error on the server.
- `record` - The record associated with `eventName`.
- `records` - The realtime, sorted records.

<code>action</code>	<code>eventName</code>	<code>record</code>	<code>records</code>	<code>source</code>	<code>description</code>
snapshot	-	-	yes	-	snapshot performed
add-listeners	-	-	yes	-	started listening to service events
mutated	see below	yes	yes	yes	record added to or mutated within publication
left-pub	see below	yes	yes	yes	mutated record is no longer within publication
remove	see below	yes	yes	yes	record within publication has been deleted
change-sort	-	-	yes	-	records resorted using the new sort criteria
remove-listeners	-	-	yes	-	stopped listening to service events

| `eventName` may be `created`, `updated`, `patched` OR `removed`.

ProTip: Two events are emitted for each optimistic mutation of the client replica. The first occurs when the client replica is mutated. It is identified by `source = 1` (see Event information below). A successful server mutation produces another event having `source = 0`. A failed server mutation reverts the record in the client replica back to its original value. That produces an event having `source = 2`.

Configure Publication

Publications

`publications` are objects containing multiple `publication` functions. These functions determine if a record belongs in the publication or not. A sample publications is:

```
const publications = {
  username: username => data => data.username === username,
  active: () => data => !data.deleted,
};
```

The publication `publications.username('john')` selects all records whose `username` is `john`; `publications.active()` selects all logically active records.

You can use the builtin `query` publication to selects records based on the [query syntax used by MongoDB](#). For example:

```
import commonPublications from 'feathers-offline-publication/lib/common-publications';
commonPublications.query({ username: 'john' })
```

Minimize service events

Once a client associates a Feathers service with

- a publications object, like the one above or `commonPublications`,
- a publication function name, and
- params for that function,

then that client will only be sent service events relevant to that publication. This may improve performance, especially for mobile devices, as the bandwidth consumed by the client is reduced.

The server-side filtering for offline-first generally needs to look at both the previous and the new contents of the record, to see if it used to belong or if it now belongs to the publication.

You can stash the current value of a record inside the hook object, before mutating it, with:

```
module.exports = {
  before: {
    update: stashBefore(),
    patch: stashBefore(),
    remove: stashBefore(),
  },
};
```

The client will receive a service event if either the previous (stashed) value of the record, or the new value is within the publication. This double check informs the client of records which previously belonged to the publication, but no longer do so after the mutation.

When records remain in the same publication

It's not uncommon, for example, for mobile apps to have unique data per user. Each service model has a `username` field and, once that field is set on `create`, it never changes.

The client would use a publication such as the `publications.username('john')` from above to select only the records for its user.

There is no need in this case to check the previous (stashed) value of the record, and you can eliminate doing so by not running the `stashBefore` hook. This would also marginally improve performance since `stashBefore` makes a `get` call.

Example

On server:

```
const serverPublication = require('feathers-offline-publication/lib/server');
const commonPublications = require('feathers-offline-publication/lib/common-publications');
const app = feathers()...

const port = app.get('port');
const server = app.listen(port);

// Configure service event filters for 2 services
serverPublication(app, commonPublications, ['messages', 'channels']);
```

ProTip: `serverPublication` must be called after the server starts listening.

On client:

```
const Realtime = require('feathers-offline-realtime');
const clientPublication = require('feathers-offline-publication/lib/client');
const commonPublications = require('feathers-offline-publication/lib/common-publications');
const feathersClient = feathers()...

const messages = feathersClient.service('messages');
const username = 'john';

// The only service events to arrive will be those relevant to the publication
messages.on('created', data => ...);
messages.on('updated', data => ...);
messages.on('patched', data => ...);
messages.on('remove', data => ...);

// Configure the publication
const messagesPublication = clientPublication.addPublication(feathersClient, 'messages', {
  module: commonPublications,
  name: 'query',
  params: { username },
});

// The publication's filter function is also available on the client
console.log(messagesPublication({ username: 'john' })); // true
console.log(messagesPublication({ username: 'jack' })); // false

// Configure the replicator
const messagesRealtime = new Realtime(messages, { publication: messagesPublication });
```

Note that the same `publications` object must be provided both on the server and the client. Also note the client may use the resultant function for any of its own filtering.

Security

An attacker may modify the `clientPublication.addPublication` call on the client or issue one of their own.

Feathers supports multiple service events filters for a method, and a mutation must satisfy them all before being emitted to the client. You can therefore add filters both before and after the `serverPublication` call to establish any additional security you need.

Installation

```
npm install feathers-offline-publication --save
```

Documentation

`serverPublication(app, publications, ...serviceNames)`

Configures services on the server which may have publications. This also configures the service event filters for you.

Options:

- `app` (*required*) - The Feathers server app.
- `publications` (*required*, object) - The publications object. The same object must be used in `clientPublication.addPublication`.
- `serviceNames` (*required*, string or array of strings) - The service name or names to configure for publications.

`clientPublication.addPublication(clientApp, serviceName, options)`

Configures a publication on the client for a remote service.

Options:

- `clientApp` (*required*) - The Feathers client app.
- `serviceName` (*required*, string) - The service name for which a publication is being configured.
- `options` (*required*, objects) - Contains
 - `module` (*required*, object) - The publications object. The same object must be used in `serverPublication`.
 - `name` (*required*, string) - The prop name of the publication in `module`.
 - `params` (*optional*, any or array of any) - The parameters to call `name` with.
 - `ifServer` (*optional*, boolean, default true) - If false, no server publication is created, but the selector function is still returned to the client.

`clientPublication.removePublication(clientApp, serviceName)`

Removes the publication for a remote service, and stops filtering on the server.

ProTip: The client will receive service events for all mutations.

Options:

- `clientApp` (*required*) - The Feathers client app.
- `serviceName` (*required*, string) - The service name whose publication is being removed.

commonPublications.query(selection)

A publication which selects records based on the [query syntax used by MongoDB](#).

Options:

- **selection** (*required*) - The [query object](#).
 - Supported operators: \$in, \$nin, \$exists, \$gte, \$gt, \$lte, \$lt, \$eq, \$ne, \$mod, \$all, \$and, \$or, \$nor, \$not, \$size, \$type, \$regex, \$where, \$elemMatch
 - Regexp searches
 - Function filtering
 - sub object searching
 - dot notation searching
 - Custom Expressions
 - filtering of immutable data structures

ProTip: You can merge these common publications with your own ones using `Object.assign({}, commonPublications, myCustomPublications)`.

Examples of snapshot replication

Let's look at some snapshot replications.

Example

You can run this example with

```
cd path/to/feathers-docs/examples/offline
npm install
cd ./snapshot
npm run build
npm start
```

Then point a browser at `localhost:3030` and look at the log on the browser console.

You can see the client source [here](#), [here](#) and [here](#).

Snapshot the entire collection

The remote service data is

```
===== Read stockRemote service directly
{dept: "a", stock: "a1", _id: "LANJQx24cg9Jy2Hr"}
{dept: "a", stock: "a2", _id: "rTk5oK6gLUpjbsyP"}
{dept: "a", stock: "a3", _id: "n0xWx970Kzp89qA9"}
{dept: "a", stock: "a4", _id: "xz0wo8g671sdejg5"}
{dept: "a", stock: "a5", _id: "nEYr6D9YcQ0Ln9nW"}
{dept: "b", stock: "b1", _id: "qY8LhzTKLw4G21d6"}
{dept: "b", stock: "b2", _id: "DBQ93zzWPNiFq6wd"}
{dept: "b", stock: "b3", _id: "xyVeDj7BRXJlPo9F"}
{dept: "b", stock: "b4", _id: "mZddKB7THrbv8dHV"}
{dept: "b", stock: "b5", _id: "4RQ5BL8PHER5DEGX"}
```

Snapshot all of it.

```
const snapshot = require('feathers-offline-snapshot');
const stockRemote = app.service('/stock');

snapshot(stockRemote)
  .then(records => console.log(records));
```

```
===== snapshot, all records
{dept: "a", stock: "a1", _id: "LANJQx24cg9Jy2Hr"}
{dept: "a", stock: "a2", _id: "rTk5oK6gLUpjbsyP"}
{dept: "a", stock: "a3", _id: "n0xWx970Kzp89qA9"}
{dept: "a", stock: "a4", _id: "xz0wo8g671sdejg5"}
{dept: "a", stock: "a5", _id: "nEYr6D9YcQ0Ln9nW"}
{dept: "b", stock: "b1", _id: "qY8LhzTKLw4G21d6"}
{dept: "b", stock: "b2", _id: "DBQ93zzWPNiFq6wd"}
{dept: "b", stock: "b3", _id: "xyVeDj7BRXJlPo9F"}
{dept: "b", stock: "b4", _id: "mZddKB7THrbv8dHV"}
{dept: "b", stock: "b5", _id: "4RQ5BL8PHER5DEGX"}
```

Snapshot part of the collection

```
const snapshot = require('feathers-offline-snapshot');
const stockRemote = app.service('/stock');

snapshot(stockRemote, { dept: 'a', $sort: { stock: 1 } })
  .then(records => console.log(records));
```

```
===== snapshot, dept: 'a'
{dept: "a", stock: "a1", _id: "LANJQx24cg9Jy2Hr"}
{dept: "a", stock: "a2", _id: "rTk5oK6gLupjbsyP"}
{dept: "a", stock: "a3", _id: "n0xWx97QKzp89qA9"}
{dept: "a", stock: "a4", _id: "xz0wo8g671sdejg5"}
{dept: "a", stock: "a5", _id: "nEYr6D9YcQ0Ln9nW"}
===== Example finished.
```

Examples of realtime replication

Realtime starts with a snapshot of the remote service data. Subsequent data changes made at the remote are delivered to the client as they occur in near real time. The data changes are applied at the client in the same order as they occurred at the remote.

Replication stops when communication is lost with the server. It can be restarted on reconnection.

Example 1 - All the remote service data

Running the example

Let's see how mutations made on the server are handled by realtime replication, along with disconnections and reconnections.

You can run this example with:

```
cd path/to/feathers-mobile/examples
npm install
cd ./realtime-1
npm run build
npm start
```

Then point a browser at `localhost:3030` and look at the log on the browser console.

You can see the client source [here](#), [here](#) and [here](#).

Looking at the log

Configure the replication and start it:

```
import Realtime from 'feathers-offline-realtime';
const stockRemote = feathersApp.service('/stock');

const stockRealtime = new Realtime(stockRemote);

stockRealtime.connect().then( ... );
```

A snapshot of the remote service data is sent to the client when replication starts.

```
===== stockRemote, before mutations
{dept: "a", stock: "a1", _id: "fY6ezNH9Rlw2WVzX"}
{dept: "a", stock: "a2", _id: "7a0b00diX18W03Gm"}
{dept: "a", stock: "a3", _id: "b2wVdYJeicCNTGLc6"}
{dept: "a", stock: "a4", _id: "wtTVYE15plCOb2vW"}
{dept: "a", stock: "a5", _id: "cnWD1Yzr8WJru0fi"}
```

```
stockRealtime.store.records.forEach(record => console.log(record))
```

```
===== client replica, before mutations
{dept: "a", stock: "a2", _id: "7a0b00diX18W03Gm"}
{dept: "a", stock: "a3", _id: "b2wVdYJeicCNTGLc6"}
{dept: "a", stock: "a5", _id: "cnWD1Yzr8WJru0fi"}
```

```
{dept: "a", stock: "a1", _id: "fY6ezNH9Rlw2WvzX"}  
{dept: "a", stock: "a4", _id: "wtTVYE15plC0b2vW"}
```

We can simulate other people changing data on the remote service.

```
===== mutate stockRemote  
stockRemote.patch stock: a1  
stockRemote.create stock: a99  
stockRemote.remove stock: a2
```

The mutations are replicated to the client.

```
===== stockRemote, after mutations  
{dept: "a", stock: "a1", _id: "fY6ezNH9Rlw2WvzX", foo: 1}  
{dept: "a", stock: "a3", _id: "b2wVdYJeiCNTGLc6"}  
{dept: "a", stock: "a4", _id: "wtTVYE15plC0b2vW"}  
{dept: "a", stock: "a5", _id: "cnWD1Yzr8WJru0fi"}  
{dept: "a", stock: "a99", _id: "Yiu8R0fHQkEaGjPz"}  
===== client replica, after mutations  
{dept: "a", stock: "a3", _id: "b2wVdYJeiCNTGLc6"}  
{dept: "a", stock: "a5", _id: "cnWD1Yzr8WJru0fi"}  
{dept: "a", stock: "a4", _id: "wtTVYE15plC0b2vW"}  
{dept: "a", stock: "a1", _id: "fY6ezNH9Rlw2WvzX", foo: 1}  
{dept: "a", stock: "a99", _id: "Yiu8R0fHQkEaGjPz"}
```

We can inform the replicator of a lost connection, after which other people mutate more data.

```
stockRealtime.disconnect();
```

```
>>>> disconnection from server  
===== mutate stockRemote  
stockRemote.patch stock: a3  
stockRemote.create stock: a98  
stockRemote.remove stock: a5
```

After we inform the replicator of a reconnection, the client replica is brought up to date with a new snapshot.

```
stockRealtime.connect();
```

```
<<<< reconnected to server  
===== stockRemote, after reconnection  
{dept: "a", stock: "a1", _id: "fY6ezNH9Rlw2WvzX", foo: 1}  
{dept: "a", stock: "a3", _id: "b2wVdYJeiCNTGLc6", foo: 1}  
{dept: "a", stock: "a4", _id: "wtTVYE15plC0b2vW"}  
{dept: "a", stock: "a98", _id: "XCZorVYjeHBlwz93"}  
{dept: "a", stock: "a99", _id: "Yiu8R0fHQkEaGjPz"}  
===== client replica, after reconnection  
{dept: "a", stock: "a98", _id: "XCZorVYjeHBlwz93"}  
{dept: "a", stock: "a99", _id: "Yiu8R0fHQkEaGjPz"}  
{dept: "a", stock: "a3", _id: "b2wVdYJeiCNTGLc6", foo: 1}  
{dept: "a", stock: "a1", _id: "fY6ezNH9Rlw2WvzX", foo: 1}  
{dept: "a", stock: "a4", _id: "wtTVYE15plC0b2vW"}  
===== Example finished.
```

Example 2 - Selected remote service data

Running the example

Let's see how a filter function (not a "publication") allows you to replicate a selection of the remote service data.

All service events are sent to the client as no "publication" is used

You can run this example with:

```
cd path/to/feathers-mobile/examples
npm install
cd ./realtime-2
npm run build
npm start
```

Then point a browser at `localhost:3030` and look at the log on the browser console.

You can see the client source [here](#), and [here](#).

Looking at the log

The client replica will contain those records with `record.dept === 'a'`. All service events are sent to the client because a filter function is used, not a "publication". Filter functions run on the client only, while a "publication" runs both on the server (to minimize the service events sent the client) and on the client.

Configure the replication and start it:

```
import Realtime from 'feathers-offline-realtime';

const stockRemote = feathersApp.service('/stock');
stockRemote.on('patched', record => console.log(`.service event. patched`, record));

const stockRealtime = new Realtime(stockRemote, {
  publication: record => record.dept === 'a', // this is a filter func, not a "publication"
  sort: Realtime.sort('stock'), // sort the client replica
  query: { dept: 'a' }, // makes snapshots more efficient
  subscriber // logs replicator events
});

stockRealtime.connect().then( ... );

function subscriber(records, { action, eventName, source }) {
  console.log(`.replicator event. action=${action} eventName=${eventName} source=${source}`);
}
```

A snapshot of part of the remote service data is sent to the client when replication starts.

```
.replicator event. action=snapshot eventName=undefined source=undefined undefined
.replicator event. action=add-listeners eventName=undefined source=undefined undefined
===== stockRemote, before mutations
{dept: "a", stock: "a1", _id: "1wKU5HpWnumm51wK"}
{dept: "a", stock: "a2", _id: "xC2ZVq6xaUpJ0Bgb"}
{dept: "a", stock: "a3", _id: "Z0Y16Pn8d3RA7rXU"}
{dept: "a", stock: "a4", _id: "kfwCtTo1p2cpN9oN"}
{dept: "a", stock: "a5", _id: "Jx1D78Jv6S5uZHvD"}
{dept: "b", stock: "b1", _id: "meldJsoQSM80msJM"}
{dept: "b", stock: "b2", _id: "iuwjwY33XVFIjLm0U"}
{dept: "b", stock: "b3", _id: "Pws5I5A8a3dc7yyJ"}
{dept: "b", stock: "b4", _id: "n4R9UxQQxR4HMBf1"}
{dept: "b", stock: "b5", _id: "lpFPGhIIInYba698P"}
```

```
stockRealtime.store.records.forEach(record => console.log(record))
```

```
===== client replica of dept: a, before mutations
{dept: "a", stock: "a1", _id: "lwKU5HpWnumm51wK"}
{dept: "a", stock: "a2", _id: "xC2ZVq6xaUpJOBgb"}
{dept: "a", stock: "a3", _id: "Z0Y16Pn8d3RA7rXU"}
{dept: "a", stock: "a4", _id: "kfwCtTo1p2cpN9oN"}
{dept: "a", stock: "a5", _id: "Jx1D78JV6S5uZHvD"}
```

We can simulate other people changing data on the remote service.

```
===== mutate stockRemote
stockRemote.patch stock: a1 move to dept: b
stockRemote.patch stock: b1 move to dept: a
.service event. patched
{dept: "b", stock: "a1", _id: "raBDpgjM4ilKa0PX"}
.replicator event. action=left-pub eventName=patched source=0
{dept: "b", stock: "a1", _id: "raBDpgjM4ilKa0PX"}
.service event. patched
{dept: "a", stock: "b1", _id: "RKbbo7EgaAeWqqnu"}
.replicator event. action=mutated eventName=patched source=0
{dept: "a", stock: "b1", _id: "RKbbo7EgaAeWqqnu"}
===== patch some stockRemote records without changing their contents
.service event. patched
{dept: "a", stock: "a2", _id: "r9VPWIdwZsYNEAvI"}
.replicator event. action=mutated eventName=patched source=0
{dept: "a", stock: "a2", _id: "r9VPWIdwZsYNEAvI"}
.service event. patched
{dept: "a", stock: "a3", _id: "n66vXg1lBuh3XX40"}
.replicator event. action=mutated eventName=patched source=0
{dept: "a", stock: "a3", _id: "n66vXg1lBuh3XX40"}
.service event. patched
{dept: "b", stock: "b2", _id: "77rLKbncUcx0FHJM"}
.service event. patched
{dept: "b", stock: "b3", _id: "1IIQqVn8TYcopdz1"}
.service event. patched
{dept: "b", stock: "b4", _id: "rsFC19WXFTi7oa6N"}
.service event. patched
{dept: "b", stock: "b5", _id: "LqXhzpPLidkIVGuG"}
```

Notice that the last 4 service events were not relevant to our publication filter and so no replication events occurred for them. **There was no need to send these 4 service events to the client.**

The mutations are replicated to the client.

```
===== stockRemote, after mutations
{dept: "b", stock: "a1", _id: "lwKU5HpWnumm51wK"}
{dept: "a", stock: "a2", _id: "xC2ZVq6xaUpJOBgb"}
{dept: "a", stock: "a3", _id: "Z0Y16Pn8d3RA7rXU"}
{dept: "a", stock: "a4", _id: "kfwCtTo1p2cpN9oN"}
{dept: "a", stock: "a5", _id: "Jx1D78JV6S5uZHvD"}
{dept: "a", stock: "b1", _id: "meldJsoQSM80mSJm"}
{dept: "b", stock: "b2", _id: "iujwY33XVF1jLmOU"}
{dept: "b", stock: "b3", _id: "Pws5I5A8a3dC7yyJ"}
{dept: "b", stock: "b4", _id: "n4R9UxQQxR4HMBfI"}
{dept: "b", stock: "b5", _id: "lpFPghIIInYba698P"}
===== client replica of dept a, after mutations
{dept: "a", stock: "a2", _id: "xC2ZVq6xaUpJOBgb"}
{dept: "a", stock: "a3", _id: "Z0Y16Pn8d3RA7rXU"}
{dept: "a", stock: "a4", _id: "kfwCtTo1p2cpN9oN"}
{dept: "a", stock: "a5", _id: "Jx1D78JV6S5uZHvD"}
{dept: "a", stock: "b1", _id: "meldJsoQSM80mSJm"}
===== Example finished.
```

The `stock: 'a1'` record was removed from the client replica because it no longer satisfied the publication filter after mutation. The `stock: 'b1'` record was added as its mutation caused it to now satisfy the publication filter.

Example 3 - Using a publication

Let's redo Example 2 using a "publication" rather than a filter function. We expect the "publication" to minimize the number of service events sent to the client.

Running the example

You can run this example with:

```
cd path/to/feathers-mobile/examples
npm install
cd ./realtime-3
npm run build
npm start
```

Then point a browser at `localhost:3030` and look at the log on the browser console.

You can see the key server source [here](#).

You can see the client source [here](#), and [here](#).

The key server code

```
const serverPublication = require('feathers-offline-publication/lib/server');
const commonPublications = require('feathers-offline-publication/lib/common-publications');
const { stashBefore } = require('feathers-hooks-common');

const port = app.get('port');
const server = app.listen(port);

const stockRemote = app.service('stock');
stockRemote.hooks({
  before: {
    update: stashBefore(),
    patch: stashBefore(),
    remove: stashBefore(),
  },
});

serverPublication(app, commonPublications, 'stock');
```

The `stashBefore` hooks will stash the current value of the record into `context.params` before the record is mutated. The service filter will use the stashed value to check if the record used to belong to the publication.

The `serverPublication` call configures the service filters, in this case for the stock service. You can use syntax like `['messages', 'comments']` to configure multiple services at once.

ProTip: `serverPublication` must be run only after the server has started listening.

The key client code

```
const clientPublication = require('feathers-offline-publication/lib/client');
const commonPublications = require('feathers-offline-publication/lib/common-publications');
```

```

const Realtime = require('feathers-offline-realtime');

const stockRemote = feathersApp.service('/stock');

const stockRealtime = new Realtime(stockRemote, {
  publication: clientPublication.addPublication(feathersApp, 'stock', {
    module: commonPublications,
    name: 'query',
    params: { dept: 'a' },
  }),
});

stockRealtime.connect().then( ... );

```

The `publication` option is now a "publication" rather than a filter function. The `addPublication` emits the `name` and `params` values to the server, while returning a filter function as the actual value for `publication`. The server will start using `name` and `params` in the server-side service filters.

Looking at the log

The previous example #2 log contained

```

===== mutate stockRemote
stockRemote.patch stock: a1 move to dept: b
stockRemote.patch stock: b1 move to dept: a
.service event. patched
  {dept: "b", stock: "a1", _id: "raBDpgjM4ilKa0PX"}
.replicator event. action=left-pub eventName=patched source=0
  {dept: "b", stock: "a1", _id: "raBDpgjM4ilKa0PX"}
.service event. patched
  {dept: "a", stock: "b1", _id: "RKbb07EgaAeWqqnu"}
.replicator event. action=mutated eventName=patched source=0
  {dept: "a", stock: "b1", _id: "RKbb07EgaAeWqqnu"}
===== patch some stockRemote records without changing their contents
.service event. patched
  {dept: "a", stock: "a2", _id: "r9VPWIdwZsYNEAvI"}
.replicator event. action=mutated eventName=patched source=0
  {dept: "a", stock: "a2", _id: "r9VPWIdwZsYNEAvI"}
.service event. patched
  {dept: "a", stock: "a3", _id: "n66vXg1lBuh3XX40"}
.replicator event. action=mutated eventName=patched source=0
  {dept: "a", stock: "a3", _id: "n66vXg1lBuh3XX40"}
.service event. patched
  {dept: "b", stock: "b2", _id: "77rLKbncUcx0FHJM"}
.service event. patched
  {dept: "b", stock: "b3", _id: "1IIQqVn8TYcopdz1"}
.service event. patched
  {dept: "b", stock: "b4", _id: "rsfc19WxfTi7oa6N"}
.service event. patched
  {dept: "b", stock: "b5", _id: "LqXhzpPLidkIVGuG"}

```

We pointed out that the last 4 service events were not relevant to example #2's publication filter and so no replication events occurred for them. We also pointed out that **there was no need to send these 4 service events to the client**.

Our example #3 log contains

```

===== mutate stockRemote
stockRemote.patch stock: a1 move to dept: b
stockRemote.patch stock: b1 move to dept: a
.service event. patched
  {dept: "b", stock: "a1", _id: "qR5KKXAP0TAdCXBm"}
.replicator event. action=left-pub eventName=patched source=0

```

```
{dept: "b", stock: "a1", _id: "qR5KkXAP0TAdCXBm"}  
.service event. patched  
  {dept: "a", stock: "b1", _id: "LDtB9YvbJTHpOau2"}  
.replicator event. action=mutated eventName=patched source=0  
  {dept: "a", stock: "b1", _id: "LDtB9YvbJTHpOau2"}  
===== patch some stockRemote records without changing their contents  
.service event. patched  
  {dept: "a", stock: "a2", _id: "fmMfWefDVe9qrRt"}  
.replicator event. action=mutated eventName=patched source=0  
  {dept: "a", stock: "a2", _id: "fmMfWefDVe9qrRt"}  
.service event. patched  
  {dept: "a", stock: "a3", _id: "WULt4o6N69G9x0Vf"}  
.replicator event. action=mutated eventName=patched source=0  
  {dept: "a", stock: "a3", _id: "WULt4o6N69G9x0Vf"}
```

The example #3 does not contain the last 4 `.service event` entries. **The server-side service filters recognized those mutations were not relevant to our publication and did not emit them to the client.**

Job done!

Example using optimistic mutation

The realtime replicator's optimistic mutation may be used to produce a snappier response at the client.

Its also an important step towards allowing the client to continue working while its offline.

The optimistic-mutator service immediately updates the client replica to what it **optimistically** expects the final result will be, and **the user can see the change right away**. The replicator then emits a replication event because the client replica data has changed.

Next, the same processing occurs as for a remote service call: the call to the server, the database processing, the filter, the service event on the client. Finally the replicator replaces the optimistic copy of the record with the one provided by the server. Then replicator emits another event because the client replica (may have) changed.

But what happens if the remote service rejects the mutation with an error? The replicator has kept a copy of the record from before the mutation and, once it detects the error response, it replaces the optimistic copy of the record with the prior version. The replicator emits a different event when this happen.

Let's take the realtime example #1 and refactor it for optimistic mutation.

[Realtime example #1](#) mutated data by calling methods on the remote service located on the server. The client had to wait until the server finished the call and until it received the service event. Only then could it mutate the client replica.

Let's refactor the client so that it instead makes those same mutations by calling the optimistic-mutator at the client. The optimistic-mutator will immediately mutate the client replica. It will then call the server and, after some delay, process the service event.

Running the example

You can run this example with:

```
cd path/to/feathers-mobile/examples
npm install
cd ./optimistic
npm run build
npm start
```

Then point a browser at `localhost:3030` and look at the log on the browser console.

You can see the client source [here](#), and [here](#).

Looking at the log

We configure the replication on the client and start it:

```
import Realtime from 'feathers-offline-realtime';

const stockRemote = feathersApp.service('/stock');
stockRemote.on('created', record => console.log(`.service event. created`, record));
stockRemote.on('updated', record => console.log(`.service event. updated`, record));
stockRemote.on('patched', record => console.log(`.service event. patched`, record));
stockRemote.on('removed', record => console.log(`.service event. removed`, record));

const stockRealtime = new Realtime(stockRemote, { uuid: true, subscriber });

feathersApp.use('stockClient', optimisticMutator({ replicator: stockRealtime }));
const stockClient = feathersApp.service('stockClient');
```

```
stockRealtime.connect().then( ... );

function subscriber(records, { action, eventName, source, record }) {
  console.log(`replicator event. action=${action} eventName=${eventName} source=${source}`, record);
}
```

A snapshot of the remote service data is sent to the client when replication starts.

```
.replicator event. action=snapshot eventName=undefined source=undefined undefined
.replicator event. action=add-listeners eventName=undefined source=undefined undefined
===== stockRemote, before mutations
{dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2"}
{dept: "a", stock: "a2", uuid: "a2", _id: "XhnvXIvFWegBRH3G"}
{dept: "a", stock: "a3", uuid: "a3", _id: "WcaLplDzLmQYdX1E"}
{dept: "a", stock: "a4", uuid: "a4", _id: "xEvdEXB1T0zJ9HB8"}
{dept: "a", stock: "a5", uuid: "a5", _id: "oDMhpBWCfQAg1Hbz"}
```

```
stockClient.find()
.then(result => console.log(result.data || result));
```

ProTip: The `find(data, params)` and `get(uuid, params)` methods of the optimistic mutator are the preferred ways to obtain data from the client replica.

```
===== client replica, before mutations
{dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2"}
{dept: "a", stock: "a2", uuid: "a2", _id: "XhnvXIvFWegBRH3G"}
{dept: "a", stock: "a3", uuid: "a3", _id: "WcaLplDzLmQYdX1E"}
{dept: "a", stock: "a4", uuid: "a4", _id: "xEvdEXB1T0zJ9HB8"}
{dept: "a", stock: "a5", uuid: "a5", _id: "oDMhpBWCfQAg1Hbz"}
```

We mutate the data with the optimistic-mutator

```
console.log('===== mutate stockRemote')
console.log('stockRemote.patch stock: a1')
stockClient.patch('a1', { foo: 1 })
.then(() => console.log('stockRemote.create stock: a99'))
.then(() => stockClient.create({ dept: 'a', stock: 'a99', uuid: 'a99' }))
.then(() => console.log('stockRemote.remove stock: a2'))
.then(() => stockClient.remove('a2'))
```

```
===== mutate stockRemote
stockRemote.patch stock: a1
.replicator event. action=mutated eventName=patched source=1
  {dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2", foo: 1}
stockRemote.create stock: a99
.replicator event. action=mutated eventName=created source=1
  {dept: "a", stock: "a99", uuid: "a99"}
stockRemote.remove stock: a2
.replicator event. action=remove eventName=removed source=1
  {dept: "a", stock: "a2", uuid: "a2", _id: "XhnvXIvFWegBRH3G"}
.service event. patched
  {dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2", foo: 1}
.replicator event. action=mutated eventName=patched source=0
  {dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2", foo: 1}
.service event. created
  {dept: "a", stock: "a99", uuid: "a99", _id: "t1FJB9f6mPZS21K5"}
.replicator event. action=mutated eventName=created source=0
  {dept: "a", stock: "a99", uuid: "a99", _id: "t1FJB9f6mPZS21K5"}
.service event. removed
  {dept: "a", stock: "a2", uuid: "a2", _id: "XhnvXIvFWegBRH3G"}
```

```
.replicator event. action=remove eventName=removed source=0
  {dept: "a", stock: "a2", uuid: "a2", _id: "XhnvXIvFWegBRH3G"}
```

You can see the replicator's optimistic mutate events `.replicator event ... source=1` occur right after the service call. That's because the client replica is being mutated immediately.

You then see the service events `.service event. patched` as the server responds to the calls made to it. This is followed by the replicator's `.replicator event ... source=0` as it processes the service event.

The client replica is immediately mutated. The matching service event was handled when it arrived later.

The client replica remain synchronised with the server data.

```
===== stockRemote, after mutations
{dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2", foo: 1}
{dept: "a", stock: "a3", uuid: "a3", _id: "WcaLplDzLmQYdX1E"}
{dept: "a", stock: "a4", uuid: "a4", _id: "xEVdEXB1T0zJ9HB8"}
{dept: "a", stock: "a5", uuid: "a5", _id: "oDMhPbWCfQAg1Hbz"}
{dept: "a", stock: "a99", uuid: "a99", _id: "t1FJB9f6mPZS21K5"}
===== client replica, after mutations
{dept: "a", stock: "a3", uuid: "a3", _id: "WcaLplDzLmQYdX1E"}
{dept: "a", stock: "a5", uuid: "a5", _id: "oDMhPbWCfQAg1Hbz"}
{dept: "a", stock: "a4", uuid: "a4", _id: "xEVdEXB1T0zJ9HB8"}
{dept: "a", stock: "a99", uuid: "a99"}
{dept: "a", stock: "a1", uuid: "a1", _id: "AHjkPcl0Kcf25xy2", foo: 1}
{dept: "a", stock: "a99", uuid: "a99", _id: "t1FJB9f6mPZS21K5"}
===== Example finished.
```

It works!

Tests as Examples

The tests in each repo can, unsurprisingly, be a valuable source of information, especially about details.

I find it frankly amazing that one Mocha test module can act as both the server and the client. You configure a Feathers server the normal way and have it listen to, say, `localhost:3030`. You then configure a Feathers WebSockets client the normal way and have it connect to that same url. Because of Feathers' design, the code runs in exactly the same way as if the server and client were on separate platforms. Very elegant.

This capability allows us to code integration tests, testing end to end, within one module. You will see this design being used in the more complicated tests, e.g. `feathers-offline-publication`.

Snapshot

- non-paginated service
- paginated service
- selection

Realtime

- [Snapshot](#)
 - query
 - publication function (not using `feathers-offline-publication`)
 - sort
 - change sort order
- [Service events](#)
 - no publication
 - mutations remaining within publication
 - mutations remaining outside publication
 - mutations moving in/out of publication
- [Optimistic mutation](#)
 - throws when not connected
 - no publication
 - no publication, `id` is null
 - no publication, remote service returns error

Publication

- [adds, removes publication](#)
- [filtering](#)

More example

More examples will soon be available.

Advanced guides

In this section you can find some guides for advanced topics once you [learned the basics](#) and created your first app.

- [Debugging](#)
- [Configuration](#)
- [File uploads](#)
- [Creating a Feathers plugin](#)
- [Seeding services](#)
- [Using a view engine](#)
- [Scaling](#)

Debugging your Feathers app

You can debug your Feathers app the same as you would any other Node app. There are [a few different options](#) you can resort to. NodeJS has a [built in debugger](#) that works really well by simply running:

```
node debug src/
```

Debugging with Visual Studio Code

Debugging Feathers with Visual Studio Code

Learn how to setup an excellent, free debugger for your Feathers application. This guide also covers some basics for those who are new to debugging and/or Feathers.

Moar Logs!

In addition to setting breakpoints we also use the fabulous `debug` module throughout Feathers core and many of the plug-ins. This allows you to get visibility into what is happening inside all of Feathers by simply setting a `DEBUG` environmental variable to the scope of modules that you want visibility into.

- Debug logs for all the things

```
DEBUG=* npm start
```

- Debug logs for all Feathers modules

```
DEBUG=feathers* npm start
```

- Debug logs for a specific module

```
DEBUG=feathers-authentication* npm start
```

- Debug logs for a specific part of a module

```
DEBUG=feathers-authentication:middleware npm start
```

Using Hooks

Since `hooks` can be registered dynamically anywhere in your app, using them to debug your state at any point in the hook chain (either before or after a service call) is really handy. For example,

```
const hooks = require('feathers-authentication').hooks;

const myDebugHook = function(hook) {
  // check to see what is in my hook object after
  // the token was verified.
  console.log(hook);
};

// Must be logged in do anything with messages.
```

```
app.service('messages').before({
  all: [
    hooks.verifyToken(),
    myDebugHook,
    hooks.populateUser(),
    hooks.restrictToAuthenticated()
  ]
});
```

You can then move that hook around the hook chain and inspect what your `hook` object looks like.

App Configuration

`feathers-configuration` allows you to load default and environment specific JSON and/or JS configuration files and environment variables and set them on your application. In a [generated application](#) the `config/default.json` and `config/production.json` files are set up with `feathers-configuration` automatically.

Here is what it does:

- Given a `NODE_CONFIG_DIR` environment variable it will load a `default.json` in that path, the default here is `./config`.
- When the `NODE_ENV` is not `development`, also try to load `<NODE_ENV>.json` in that path and merge both configurations (with `<NODE_ENV>.json` taking precedence)
- Go through each configuration value and sets it on the application (via `app.set(name, value)`).
 - If the value is a valid environment variable (e.v. `NODE_ENV`), use its value instead
 - If the value start with `./` or `../` turn it to an absolute path relative to the configuration file path
 - If the value starts with a `\`, do none of the above two

Usage

The `feathers-configuration` module is an app configuration function that takes a root directory (usually something like `__dirname` in your application) and the configuration folder (set to `config` by default):

```
const feathers = require('feathers');
const configuration = require('feathers-configuration')

// Use the current folder as the root and look configuration up in `settings`
const app = feathers().configure(configuration(__dirname, 'settings'))
```

Example

In `config/default.json` we want to use the local development environment and default MongoDB connection string:

```
{
  "frontend": "../public",
  "host": "localhost",
  "port": 3030,
  "mongodb": "mongodb://localhost:27017/myapp",
  "templates": "../templates"
}
```

In `config/production.js` we are going to use environment variables (e.g. set by Heroku) and use `public/dist` to load the frontend production build:

```
module.exports = {
  "frontend": "./public/dist",
  "host": "myapp.com",
  "port": "PORT",
  "mongodb": "MONGOHQ_URL"
}
```

Now it can be used in our `app.js` like this:

```

const feathers = require('feathers');
const configuration = require('feathers-configuration')

const app = feathers()
  .configure(configuration(__dirname));

console.log(app.get('frontend'));
console.log(app.get('host'));
console.log(app.get('port'));
console.log(app.get('mongodb'));
console.log(app.get('templates'));

```

If you now run

```

node app
// -> path/to/app/public
// -> localhost
// -> 3030
// -> mongodb://localhost:27017/myapp
// -> path/to/templates

```

Or via custom environment variables by setting them in `config/custom-environment-variables.json`:

```
{
  "port": "PORT",
  "mongodb": "MONGOHQ_URL"
}
```

```

PORT=8080 MONGOHQ_URL=mongodb://localhost:27017/production NODE_ENV=production node app
// -> path/to/app/public/dist
// -> myapp.com
// -> 8080
// -> mongodb://localhost:27017/production
// -> path/to/templates

```

You can prevent interpolation of environment variables by prefacing the value with \

```

"authentication": {
  "local": {
    "entity": "\\\USER",
    "usernameField": "\\\USERNAME",
  }
}

```

You can also override these variables with arguments. Read more about how with [node-config](#)

File uploads in FeathersJS

Over the last months we at ciancoders.com have been working in a new SPA project using Feathers and React, the combination of those two turns out to be **just amazing**.

Recently we were struggling to find a way to upload files without having to write a separate Express middleware or having to (re)write a complex Feathers service.

Our Goals

We want to implement an upload service to accomplish a few important things:

1. It has to handle large files (+10MB).
2. It needs to work with the app's authentication and authorization.
3. The files need to be validated.
4. At the moment there is no third party storage service involved, but this will change in the near future, so it has to be prepared.
5. It has to show the upload progress.

The plan is to upload the files to a feathers service so we can take advantage of hooks for authentication, authorization and validation, and for service events.

Fortunately, there exists a file storage service: [feathers-blob](#). With it we can meet our goals, but (spoiler alert) it isn't an ideal solution. We discuss some of its problems below.

Basic upload with feathers-blob and feathers-client

For the sake of simplicity, we will be working over a very basic feathers server, with just the upload service.

Lets look at the server code:

```
/* --- server.js --- */

const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const hooks = require('feathers-hooks');
const bodyParser = require('body-parser');
const handler = require('feathers-errors/handler');

// feathers-blob service
const blobService = require('feathers-blob');
// Here we initialize a FileSystem storage,
// but you can use feathers-blob with any other
// storage service like AWS or Google Drive.
const fs = require('fs-blob-store');
const blobStorage = fs(__dirname + '/uploads');

// Feathers app
const app = feathers();

// Parse HTTP JSON bodies
app.use(bodyParser.json());
// Parse URL-encoded params
app.use(bodyParser.urlencoded({ extended: true }));
```

```
// Register hooks module
app.configure(hooks());
// Add REST API support
app.configure(rest());
// Configure Socket.io real-time APIs
app.configure(socketio());

// Upload Service
app.use('/uploads', blobService({Model: blobStorage}));

// Register a nicer error handler than the default Express one
app.use(handler());

// Start the server
app.listen(3030, function(){
  console.log('Feathers app started at localhost:3030')
});
```

`feathers-blob` works over `abstract-blob-store`, which is an abstract interface to various storage backends, such as filesystem, AWS, or Google Drive. It only accepts and retrieves files encoded as `dataURI` strings.

Just like that we have our backend ready, go ahead and POST something to `localhost:3030/uploads``, for example with postman:

```
{
  "uri": "data:image/gif;base64,R0lGODlhEwATAPcAAP/+//7///+///fvzYvryYvvzz/fxg/zxWfvxW/zwXPrLw/vxXvfrXv3xY
vrvYntvnyv/ruzPrwZPfsZPjszfjtZvfsZvHmY/zxavftaPrvavjuafzxbfnua/jta/ftbP3yb/zccPvb/zccfvxctzxc/3zdf3zdv70efvw
d/rwd/vwefft3/3yfPxvfP70f/zzfvnwffvzf/rxf/rxgPjvgPjvgfnwhPvzhvjvhv71jfz0kPrykvz0mv72nvblTPnnUPjoUPrpUvnvUfnpuVx
1Ufnpu/nPVnqVPfnU/3uvvsvWPfpVvnqWfrXPLiW/nrX/vtYv7xavrta/Hlcvnuf/Pphvbsif3zk/zzlPzylfjuk/z0o/LqnrbhSPbhSfjis/
j1S/jjTPfhTfjlTubUU+iPPokpkrvl/D11/ftovLWPfHXPvHZP/PbQ/bcRuDJP/PaRvjgSffdSe3ddu7fge7fi+zku07NMvPT0t2/Nu7S0+300
/PWQdnGbOneqeqvDqyu3JMuvJMv7KNFHNON7GZdnEbejanObXn0W8J0a9K0vCLOnBK9+4Ku3FL9ayKuzEMcenK9e+X0D0iePSk0D0k0W3Itis
I9yxL+a9NtGiHr+VH5h5JssfNM2bGN6rMj4JMOYL5h4JZ15Jph3Jp14J5h5Jh3KJ14KzP5Ks+sUN7G1961LL+PKMmbMzt2Jpp3Jpt3KZ14K7q
FFdyiKdufKsedRdm7feOpQN2QKMKENRpVjBFFIrNjJL1mLMBpLr9oLrFhK69bJFkpE1kpFYNeTqFEilsoFbmlnlsmFFwpGFkoF///7+/v///w
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACHSBAEAANALAATAABMAAAj/AKEJHCgo
KKJKlhThGciQYSIVa7r8SHPFzqGGAwPd4bKlh5YsPKy0qFLnT0NAaHTcsIHdho0aKkaAwGCGEKm1NmSkIjWLBoSVJT6c0jUrzsBKP154KmYsACo
TMmk1WwaA1CRoeM7siJEqmTAsjp40ICK2bEApfZcsQlxwxRzgI8w8hgrovYA+Kq6sMK0QEKVCKUkoVqQwQJFTwFEAAAFZ9p1Fy40EE1IYJD5
5EodDA1C1TbPp0okRFxQDBRgskAKhiRM1c+Ss4SnPFCIoBBwkUMBkCBIiY8qAqcPGOKBhrBTFQbCEV5EjQYQAcfNFjp5CxpxagVtUhIjwzaJY
SHzhQ4cP3ryQHLEqJbASnu+6EIw6o2b2X0ISXK0CFSugazs0YYmwQhziyuE2PLLIv3h0hArkRhiccZAEONLL7tgAoqDGLXSSSaPMLIIJpmAUst/
GA3UCiuv1PIKLtw1FBAA0w=="
}
```

The service will respond with something like this:

```
{
  "id": "6454364d8facd7a88e627e4c4b11b032d2f83af8f7f9329ffc2b7a5c879dc838.gif",
  "uri": "the-same-uri-we-uploaded",
  "size": 1156
}
```

Or we can implement a very basic frontend with `feathers-client` and `jquery`:

```
<!doctype html>
<html>
  <head>
    <title>Feathersjs File Upload</title>
    <script src="https://code.jquery.com/jquery-2.2.3.min.js" integrity="sha256-a23g1Nt4dtEY0j7bR+vTu7+T8VP13humZFBJN1YoEJo=" crossorigin="anonymous"></script>
    <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
  >
  <script type="text/javascript" src="//unpkg.com/feathers-client@^2.0.0/dist/feathers.js"></script>
  <script type="text/javascript">
```

```

// feathers client initialization
const rest = feathers().rest('http://localhost:3030');
const app = feathers()
  .configure(feathers.hooks())
  .configure(rest.jquery($));

// setup jQuery to watch the ajax progress
$.ajaxSetup({
  xhr: function () {
    var xhr = new window.XMLHttpRequest();
    // upload progress
    xhr.addEventListener("progress", function (evt) {
      if (evt.lengthComputable) {
        var percentComplete = evt.loaded / evt.total;
        console.log('upload progress: ', Math.round(percentComplete * 100) + "%");
      }
    }, false);
    return xhr;
  }
});

const uploadService = app.service('uploads');
const reader = new FileReader();

// encode selected files
$(document).ready(function(){
  $('#file').change(function(){
    var file = this.files[0];
    // encode dataURI
    reader.readAsDataURL(file);
  })
});

// when encoded, upload
reader.addEventListener("load", function () {
  console.log('encoded file: ', reader.result);
  var upload = uploadService
    .create({uri: reader.result})
    .then(function(response){
      // success
      alert('UPLOADED!! ');
      console.log('Server responded with: ', response);
    });
}, false);
</script>
</head>
<body>
  <h1>Let's upload some files!</h1>
  <input type="file" id="file"/>
</body>
</html>

```

This code watches for file selection, then encodes it and does an ajax post to upload it, watching the upload progress via the xhr object. Everything works as expected.

Every file we select gets uploaded and saved to the `./uploads` directory.

Work done!, let's call it a day, shall we?

... But hey, there is something that doesn't feels quite right ...right?

DataURI upload problems

It doesn't feel right because it is not. Let's imagine what would happen if we try to upload a large file, say 25MB or more: The entire file (plus some extra MB due to the encoding) has to be kept in memory for the entire upload process, this could look like nothing for a normal computer but for mobile devices it's a big deal.

We have a big RAM consumption problem. Not to mention we have to encode the file before sending it...

The solution would be to modify the service, adding support for splitting the dataURI into small chunks, then uploading one at a time, collecting and reassembling everything on the server. But hey, it's not that the same thing browsers and web servers have been doing since maybe the very early days of the web? maybe since Netscape Navigator?

Well, actually it is, and doing a `multipart/form-data` post is still the easiest way to upload a file.

Feathers-blob with multipart support.

Back with the backend, in order to accept multipart uploads, we need a way to handle the `multipart/form-data` received by the web server. Given that Feathers behaves like Express, let's just use `multer` and a custom middleware to handle that.

```
/* --- server.js --- */
const multer = require('multer');
const multipartMiddleware = multer();

// Upload Service with multipart support
app.use('/uploads',

    // multer parses the file named 'uri'.
    // Without extra params the data is
    // temporarily kept in memory
    multipartMiddleware.single('uri'),

    // another middleware, this time to
    // transfer the received file to feathers
    function(req,res,next){
        req.feathers.file = req.file;
        next();
    },
    blobService({Model: blobStorage})
);
```

Notice we kept the file field name as *uri* just to maintain uniformity, as the service will always work with that name anyways. But you can change it if you prefer.

Feathers-blob only understands files encoded as dataURI, so we need to convert them first. Let's make a Hook for that:

```
/* --- server.js --- */
const dauria = require('dauria');

// before-create Hook to get the file (if there is any)
// and turn it into a datauri,
// transparently getting feathers-blob to work
// with multipart file uploads
app.service('/uploads').before({
    create: [
        function(hook) {
            if (!hook.data.uri && hook.params.file){
                const file = hook.params.file;
                const uri = dauria.getBase64DataURI(file.buffer, file.mimetype);
                hook.data = {uri: uri};
            }
        }
    ]
});
```

```
    ];
});
```

Et voilà! Now we have a FeathersJS file storage service working, with support for traditional multipart uploads, and a variety of storage options to choose.

Simply awesome.

Further improvements

The service always return the dataURI back to us, which may not be necessary as we'd just uploaded the file, also we need to validate the file and check for authorization.

All those things can be easily done with more Hooks, and that's the benefit of keeping all inside FeathersJS services. I left that to you.

For the frontend, there is a problem with the client: in order to show the upload progress it's stuck with only REST functionality and not real-time with socket.io.

The solution is to switch `feathers-client` from REST to `socket.io`, and just use wherever you like for uploading the files, that's an easy task now that we are able to do a traditional `form-multipart` upload.

Here is an example using dropzone:

```
<!doctype html>
<html>
  <head>
    <title>Feathersjs File Upload</title>

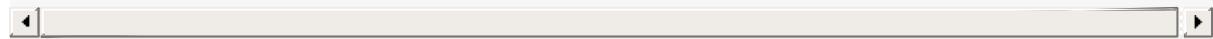
    <link rel="stylesheet" href="assets/dropzone.css">
    <script src="assets/dropzone.js"></script>

    <script type="text/javascript" src="socket.io/socket.io.js"></script>
    <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
  >
    <script type="text/javascript" src="//unpkg.com/feathers-client@^2.0.0/dist/feathers.js"></script>
    <script type="text/javascript">
      // feathers client initialization
      var socket = io('http://localhost:3030');
      const app = feathers()
        .configure(feathers.hooks())
        .configure(feathers.socketio(socket));
      const uploadService = app.service('uploads');

      // Now with Real-Time Support!
      uploadService.on('created', function(file){
        alert('Received file created event!', file);
      });

      // Let's use DropZone!
      Dropzone.options.myAwesomeDropzone = {
        paramName: "uri",
        uploadMultiple: false,
        init: function(){
          this.on('uploadprogress', function(file, progress){
            console.log('progress', progress);
          });
        }
      };
    </script>
  </head>
  <body>
```

```
<h1>Let's upload some files!</h1>
<form action="/uploads"
      class="dropzone"
      id="my-awesome-dropzone"></form>
</body>
</html>
```



All the code is available via github here: <https://github.com/CianCoders/feathers-example-fileupload>

Hope you have learned something today, as I learned a lot writing this.

Cheers!

Creating a Feathers Plugin

The easiest way to create a plugin is with the [Yeoman generator](#).

First install the generator

```
$ npm install -g generator-feathers-plugin
```

Then in a new directory run:

```
$ yo feathers-plugin
```

This will scaffold out everything that is needed to start writing your plugin.

Output files from generator:

```
create package.json
create .babelrc
create .editorconfig
create .jshintrc
create .travis.yml
create src/index.js
create test/index.test.js
create README.md
create LICENSE
create .gitignore
create .npmignore
```

Simple right? We technically could call it a day as we have created a Plugin. However, we probably want to do just a bit more. Generally speaking a Plugin is a [Service](#). The fun part is that a Plugin can contain multiple Services which we will create below. This example is going to build out 2 services. The first will allow us to find members of the Feathers Core Team & the second will allow us to find the best state in the United States.

Verifying our Service

Before we start writing more code we need to see that things are working.

```
$ cd example && node app.js
Error: Cannot find module '../lib/index'
```

Dang! Running the example app resulted in an error and you said to yourself, "The generator must be broken and we should head over to the friendly Slack community to start our debugging journey". Well, as nice as they may be we can get through this. Let's take a look at the package.json that came with our generator. Most notably the scripts section.

```
"scripts": {
  "prepublish": "npm run compile",
  "publish": "git push origin && git push origin --tags",
  "release:patch": "npm version patch && npm publish",
  "release:minor": "npm version minor && npm publish",
  "release:major": "npm version major && npm publish",
  "compile": "rimraf lib/ && babel -d lib/ src/",
  "watch": "babel --watch -d lib/ src/",
```

```
"jshint": "jshint src/. test/. --config",
"mocha": "mocha --recursive test/ --compilers js:babel-core/register",
"test": "npm run compile && npm run jshint && npm run mocha",
"start": "npm run compile && node example/app"
}
```

Back in business. That error message was telling us that we need to build our project. In this case it means babel needs to do its thing. For development you can run watch

```
$ npm run watch

> creatingPlugin@0.0.0 watch /Users/ajones/git/training/creatingPlugin
> babel --watch -d lib/ src/

src/index.js -> lib/index.js
```

After that you can run the example app and everything should be good to go.

```
$ node app.js
Feathers app started on 127.0.0.1:3030
```

Expanding our Plugin

Now that we did our verification we can safely change things. For this example we want 2 services to be exposed from our Plugin. Let's create a services directory within the src folder.

```
// From the src directory
$ mkdir services
$ ls
index.js services
```

Now let's create our two services. We will just copy the index.js file.

```
$ cp index.js services/core-team.js
$ cp index.js services/best-state.js
$ cd services && ls
best-state.js core-team.js

$ cat best-state.js

if (!global._babelPolyfill) { require('babel-polyfill'); }

import errors from 'feathers-errors';
import makeDebug from 'debug';

const debug = makeDebug('creatingPlugin');

class Service {
  constructor(options = {}) {
    this.options = options;
  }

  find(params) {
    return new Promise((resolve, reject) => {
      // Put some async code here.
      if (!params.query) {
        return reject(new errors.BadRequest());
      }

      resolve([]);
    });
  }
}
```

```

    });
}

export default function init(options) {
  debug('Initializing creatingPlugin plugin');
  return new Service(options);
}

init.Service = Service;

```

At this point we have index.js, best-state.js and core-team.js with identical content. The next step will be to change index.js so that it is our main file.

Our new index.js

```

if (!global._babelPolyfill) { require('babel-polyfill'); }

import coreTeam from './services/core-team';
import bestState from './services/best-state';

export default { coreTeam, bestState };

```

Now we need to actually write the services. We will only be implementing the find action as you can reference the [service docs](#) to learn more. Starting with core-team.js we want to find out the names of the members listed in the feathers.js org on github.

```

//core-team.js
if (!global._babelPolyfill) { require('babel-polyfill'); }

import errors from 'feathers-errors';
import makeDebug from 'debug';

const debug = makeDebug('creatingPlugin');

class Service {
  constructor(options = {}) {
    this.options = options;
  }

  //We are only changing the find...
  find(params) {
    return Promise.resolve(['Mikey', 'Cory Smith', 'David Luecke', 'Emmanuel Bourmalo', 'Eric Kryski',
      'Glavin Wiechert', 'Jack Guy', 'Anton Kulakov', 'Marshall Thompson']);
  }
}

export default function init(options) {
  debug('Initializing creatingPlugin plugin');
  return new Service(options);
}

init.Service = Service;

```

That will now return an array of names when service.find is called. Moving on to the best-state service we can follow the same pattern

```

if (!global._babelPolyfill) { require('babel-polyfill'); }

import errors from 'feathers-errors';
import makeDebug from 'debug';

const debug = makeDebug('creatingPlugin');

```

```

class Service {
  constructor(options = {}) {
    this.options = options;
  }

  find(params) {
    return Promise.resolve(['Alaska']);
  }
}

export default function init(options) {
  debug('Initializing creatingPlugin plugin');
  return new Service(options);
}

init.Service = Service;

```

Notice in the above service it return a single item array with the best state listed.

Usage

The easiest way to use our plugin will be within the same app.js file that we were using earlier. Let's write out some basic usage in that file.

```

//app.js
const feathers = require('feathers');
const rest = require('feathers-rest');
const hooks = require('feathers-hooks');
const bodyParser = require('body-parser');
const errorHandler = require('feathers-errors/handler');
const plugin = require('../lib/index');

// Initialize the application
const app = feathers()
  .configure(rest())
  .configure(hooks())
  // Needed for parsing bodies (login)
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  // Initialize your feathers plugin
  .use('/plugin/coreTeam', plugin.coreTeam())
  .use('/plugin/bestState', plugin.bestState())
  .use(errorHandler());

var bestStateService = app.service('/plugin/bestState')
var coreTeamService = app.service('/plugin/coreTeam')

bestStateService.find().then( (result) => {
  console.log(result)
}).catch( error => {
  console.log('Error finding the best state ', error)
})

coreTeamService.find().then( (result) => {
  console.log(result)
}).catch( error => {
  console.log('Error finding the core team ', error)
})

app.listen(3030);

console.log('Feathers app started on 127.0.0.1:3030');

```

```
$ node app.js

Feathers app started on 127.0.0.1:3030
[ 'Alaska'
[ 'Mikey',
'Cory Smith',
'David Luecke',
'Emmanuel Bourmalo',
'Eric Kryski',
'Glavin Wiechert',
'Jack Guy',
'Anton Kulakov',
'Marshall Thompson' ]
```

Testing

Our generator stubbed out some basic tests. We will remove everything and replace it with the following.

```
import { expect } from 'chai';
import plugin from '../src';

const bestStateService = plugin.bestState()

describe('bestState', () => {
  it('is Alaska', () => {
    bestStateService.find().then(result => {
      console.log(result)
      expect(result).to.eql(['Alaska']);
      done();
    });
  });
});
```

```
$ npm run test
```

Because this is just a quick sample jshint might fail. You can either fix the syntax or change the test command.

```
$ npm run compile && npm run mocha
```

This should give you the basic idea of creating a Plugin for Feathers.

Seeding Services

It is common to populate the database with mock data while developing and testing applications. This process is known as *seeding*, and the `feathers-seeder` plugin makes this easy. `feathers-seeder` seeds your *services*, so you can seed any database in the exact same way.

Installing feathers-seeder

First, install the plugin.

```
$ npm install --save feathers-seeder
```

Next, modify your `src/app.js` to look somewhat like this:

```
const feathers = require('feathers');
const seeder = require('feathers-seeder');
const seederConfig = require('./seeder-config');

const app = feathers();

app
  .configure(seeder(seederConfig));

module.exports = app;
```

Create a `src/seeder-config.js` file:

```
module.exports = {
  services: [
    {
      path: 'users',
      template: {
        name: '{{name.firstName}} {{name.lastName}}',
        password: '{{internet.password}}'
      }
    }
  ]
};
```

Lastly, in your `src/server.js`:

```
app.seed().then(() => {
  const server = app.listen(app.get('port'));
  // ...
}).catch(err => {
  // ...
});
```

`feathers-seeder` expects your configuration object to have a `services` array, where you can provide a template (which will be filled by [@marak/Faker.js](#)) that will be inserted into your service.

The configuration options are described in depth [here](#).

Seeding Nested Services

A common scenario is having a service that relies directly on another service. `feathers-seeder` allows you to include a `callback` function inside your configuration, so that you can interact with the instances you create.

For example, if you had a service called `apartments`, and another called `apartments/:apartmentId/tenants`:

```
export default const seederConfig = {
  services: [
    {
      count: 25, // Create 25 apartments
      path: 'apartments',
      template: {
        city: '{{address.city}}',
        zip: '{{address.zipCode}}'
      },
      callback(apartment, seed) {
        // Create 10 tenants for each apartment
        return seed({
          count: 10,
          path: 'apartments/:apartmentId/tenants',
          template: {
            name: '{{name.firstName}} {{name.lastName}}',
            email: '{{internet.email}}'
          },
          params: {
            apartmentId: apartment._id
          }
        });
      }
    }
  ];
};
```

Keep in mind, your callback function *must* return a `Promise`.

Again, all configuration options are listed [here](#). Happy seeding!

Using A View Engine

Since Feathers is just an extension of Express it's really simple to render templated views on the server with data from your Feathers services. There are a few different ways that you can structure your app so this guide will show you 3 typical ways you might have your Feathers app organized.

A Single "Monolithic" App

You probably already know that when you register a Feathers service, Feathers creates RESTful endpoints for that service automatically. Well, really those are just Express routes, so you can define your own as well.

ProTip: Your own defined REST endpoints won't work with hooks and won't emit socket events. If you find you need that functionality it's probably better for you to turn your endpoints into a minimal Feathers service.

Let's say you want to render a list of messages from most recent to oldest using the Jade template engine.

```
// You've set up your main Feathers app already

// Register your view engine
app.set('view engine', 'jade');

// Register your message service
app.use('/api/messages', memory());

// Inside your main Feathers app
app.get('/messages', function(req, res, next){
  // You namespace your feathers service routes so that
  // don't get route conflicts and have nice URLs.
  app.service('api/messages')
    .find({ query: {$sort: { updatedAt: -1 } } })
    .then(result => res.render('message-list', result.data))
    .catch(next);
});

});
```

Simple right? We've now rendered a list of messages. All your hooks will get triggered just like they would normally so you can use hooks to pre-filter your data and keep your template rendering routes super tight.

ProTip: If you call a Feathers service "internally" (ie. not over sockets or REST) you won't have a `hook.params.provider` attribute. This allows you to have hooks only execute when services are called externally vs. from your own code. See [bundled hooks](#) for an example.

Feathers As A Sub-App

Sometimes a better way to break up your Feathers app is to put your services into an API and mount your API as a sub-app. This is just like you would do with Express. If you do this, it's only a slight change to get data from your services.

```
// You've set up your main Feathers app already

// Register your view engine
app.set('view engine', 'jade');

// Require your configured API sub-app
const api = require('./api');
```

```
// Register your API sub app
app.use('/api', api);

app.get('/messages', function(req, res, next){
  api.service('messages')
    .find({ query: {$sort: { updatedAt: -1 } } })
    .then(result => res.render('message-list', result.data))
    .catch(next);
});
```

Not a whole lot different. Your API sub app is pretty much the same as your single app in the previous example, and your main Feathers app is just a really small wrapper that does little more than render templates.

Feathers As A Separate App

If your app starts to get a bit busier you might decide to move your API to a completely separate standalone Feathers app, maybe even on a different server. In order for both apps to talk to each other they'll need some way to make remote requests. Well, Feathers just so happens to have a [client side piece](#) that can be used on the server. This is how it works.

```
// You've set up your feathers app already

// Register your view engine
app.set('view engine', 'jade');

// Include the Feathers client modules
const client = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');

// Set up a socket connection to our remote API
const socket = io('http://api.feathersjs.com');
const api = client().configure(socketio(socket));

app.get('/messages', function(req, res, next){
  api.service('messages')
    .find({ query: {$sort: { updatedAt: -1 } } })
    .then(result => res.render('message-list', result.data))
    .catch(next);
});
```

ProTip: In the above example we set up sockets. Alternatively you could use a Feathers client [REST provider](#).

And with that, we've shown 3 different ways that you use a template engine with Feathers to render service data. If you see any issues in this guide feel free to [submit a pull request](#).

Scaling

Depending on your requirements, your feathers application may need to provide high availability. Feathers is designed to scale.

The types of transports used in a feathers application will impact the scaling configuration. For example, a feathers app that uses the `feathers-rest` adapter exclusively will require less scaling configuration because HTTP is a stateless protocol. If using websockets (a stateful protocol) through the `feathers-socketio` or `feathers-primus` adapters, configuration may be more complex to ensure websockets work properly.

Horizontal Scaling

Scaling horizontally refers to either:

- setting up a [cluster](#), or
- adding more machines to support your application

To achieve high availability, varying combinations of both strategies may be used.

Cluster configuration

[Cluster](#) support is built into core NodeJS. Since NodeJS is single threaded, clustering allows you to easily distribute application requests among multiple child processes (and multiple threads). Clustering is a good choice when running feathers in a multi-core environment.

Below is an example of adding clustering to feathers with the `feathers-socketio` provider. By default, websocket connections begin via a handshake of multiple HTTP requests and are upgraded to the websocket protocol. However, when clustering is enabled, the same worker will not process all HTTP requests for a handshake, leading to HTTP 400 errors. To ensure a successful handshake, force a single worker to process the handshake by disabling the http transport and exclusively using the `websocket` transport.

There are notable side effects to be aware of when disabling the HTTP transport for websockets. While all modern browsers support websocket connections, there is no websocket support for [IE <=9](#) and [Android Browser <=4.3](#). If you must support these browsers, use alternative scaling strategies.

```
import cluster from 'cluster';
import feathers from 'feathers';
import socketio from 'feathers-socketio';

const CLUSTER_COUNT = 4;

if (cluster.isMaster) {
  for (let i = 0; i < CLUSTER_COUNT; i++) {
    cluster.fork();
  }
} else {
  const app = feathers();
  // ensure the same worker handles websocket connections
  app.configure(socketio({
    transports: ['websocket']
  }));
  app.listen(4000);
}
```

In your feathers client code, limit the socket.io-client to the `websocket` transport and disable `upgrade`.

```
import hooks from 'feathers-hooks';
import feathers from 'feathers/client';
import io from 'socket.io-client';
import socketio from 'feathers-socketio/client';

const app = feathers()
  .configure(hooks())
  .configure(socketio(
    io('http://api.feathersjs.com', {
      transports: ['websocket'],
      upgrade: false
    })
  ));

```

API

- Core
 - [Application](#)
 - [Services](#)
 - [Hooks](#)
 - [Common Hooks](#)
 - [Client](#)
 - [Events](#)
 - [Errors](#)
- Transports
 - [REST](#)
 - [Express](#)
 - [Socket.io](#)
 - [Primus](#)
- Authentication
 - [Server](#)
 - [Client](#)
 - [Local](#)
 - [JWT](#)
 - [OAuth1](#)
 - [OAuth2](#)
 - [Hooks](#)
- Databases
 - [Common API](#)
 - [Querying](#)
 - [Memory](#)
 - [NeDb](#)
 - [LocalStorage](#)
 - [MongoDB](#)
 - [Mongoose](#)
 - [Sequelize](#)
 - [Knex](#)
 - [RethinkDB](#)

Application

Star 7k npm v2.2.0 changelog .md

```
$ npm install feathers --save
```

The core `feathers` module provides the ability to initialize new Feathers application instances. Each instance allows for registration and retrieval of [services](#), plugin configuration, and getting and setting global configuration options. An initialized Feathers application is referred to as the [app object](#). The API documented on this page works both, on the server and [the client](#).

```
// To create a Feathers server application
const feathers = require('feathers');

// To create a client side application
const feathers = require('feathers/client');

const app = feathers();
```

Important: In addition to the API outlined below, a Feathers server application also provides the exact same functionality as an [Express 4](#) application. For more advanced use of Feathers, familiarity with Express is highly recommended. For the interaction between Express and Feathers, also see the [REST](#) and [Express](#) chapters.

.use(path, service)

`app.use(path, service) -> app` allows registering a [service object](#) on the `path`.

```
// Add a service.
app.use('/messages', {
  get(id) {
    return Promise.resolve({
      id,
      text: `This is the ${name} message!`
    });
  }
});
```

On the server `.use` also provides the same functionality as [Express app.use](#) if passed something other than a service object (e.g. an Express middleware or other app object).

Important: REST services are registered in the same order as any other middleware. For additional information on how services and middleware interact see the [Express chapter](#).

.service(path)

`app.service(path) -> service` returns the wrapped [service object](#) for the given path. Feathers internally creates a new object from each registered service. This means that the object returned by `app.service(path)` will provide the same methods and functionality as your original service object but also functionality added by Feathers and its plugins like [service events](#) and [additional methods](#). `path` can be the service name with or without leading and trailing slashes.

```

const messageService = app.service('messages');

messageService.get('test').then(message => console.log(message));

app.use('/my/todos', {
  create(data) {
    return Promise.resolve(data);
  }
});

const todoService = app.service('my/todos');
// todoService is an event emitter
todoService.on('created', todo =>
  console.log('Created todo', todo)
);

```

.configure(callback)

`app.configure(callback)` -> app runs a `callback` function with the application as the context (`this`). It can be used to initialize plugins or services.

```

function setupService() {
  this.use('/todos', todoService);
}

app.configure(setupService);

```

.listen(port)

`app.listen([port])` -> `HTTPServer` starts the application on the given port. It will first call the original [Express app.listen\(\[port\]\)](#), then run `app.setup(server)` (see below) with the server object and then return the server object.

`listen` does nothing on the Feathers Client.

.setup(server)

`app.setup(server)` -> app is used to initialize all services by calling each `services .setup(app, path)` method (if available). It will also use the `server` instance passed (e.g. through `http.createServer`) to set up SocketIO (if enabled) and any other provider that might require the server instance.

Normally `app.setup` will be called automatically when starting the application via `app.listen([port])` but there are cases when it needs to be called explicitly. For more information see the [Express chapter](#).

.set(name, value)

`app.set(name, value)` -> app assigns setting `name` to `value`.

.get(name)

`app.get(name)` -> `value` retrieves the setting `name`. For more information on server side Express settings see the [Express documentation](#).

```
app.set('port', 3030);
app.listen(app.get('port'));
```

.hooks(hooks)

`app.hooks(hooks)` -> `app` allows registration of application-level hooks. For more information see the [application hooks section](#).

.on(eventname, listener)

Provided by the core [NodeJS EventEmitter .on](#). Registers a `listener` method (`function(data) {}`) for the given `eventname`.

```
app.on('login', user => console.log('Logged in', user));
```

.emit(eventname, data)

Provided by the core [NodeJS EventEmitter .emit](#). Emits the event `eventname` to all event listeners.

```
app.emit('myevent', {
  message: 'Something happened'
});

app.on('myevent', data => console.log('myevent happened', data));
```

.removeListener(eventname, [listener])

Provided by the core [NodeJS EventEmitter .removeListener](#). Removes all or the given listener for `eventname`.

Services

Services are the heart of every Feathers application and JavaScript objects (or instances of [ES6 classes](#)) that implements [certain methods](#). Feathers itself will also add some [additional methods and functionality](#) to its services.

Service methods

Service methods are pre-defined [CRUD](#) methods that your service object can implement (or that has already been implemented by one of the [database adapters](#)). Below is a complete example of the Feathers *service interface*:

```
const myService = {
  find(params) {},
  get(id, params) {},
  create(data, params) {},
  update(id, data, params) {},
  patch(id, data, params) {},
  remove(id, params) {},
  setup(app, path) {}
}

app.use('/my-service', myService);
```

Or as an [ES6 class](#):

```
'use strict';

class MyService {
  find(params) {}
  get(id, params) {}
  create(data, params) {}
  update(id, data, params) {}
  patch(id, data, params) {}
  remove(id, params) {}
  setup(app, path) {}
}

app.use('/my-service', new MyService());
```

ProTip: Methods are optional, and if a method is not implemented Feathers will automatically emit a `NotImplemented` error.

Service methods have to return a [Promise](#) and have the following parameters:

- `id` - the identifier for the resource. A resource is the data identified by a unique id.
- `data` - the resource data.
- `params` - can contain any extra parameters, for example the authenticated user.

Important: `params.query` contains the query parameters from the client, either passed as URL query paramters (see the [REST](#) chapter) or through websockets (see [Socket.io](#) or [Primus](#)).

Once registered the service can be retrieved and used via [app.service\(\)](#):

```
const myService = app.service('my-service');

myService.find().then(items => console.log('.find()', items));
myService.get(1).then(item => console.log('.get(1)', item));
```

Keep in mind that services don't have to use databases. You could easily replace the database in the example with a package that uses some API, like pulling in GitHub stars or stock ticker data.

Important: This section describes the general use of service methods and how to implement them. They are already implemented by Feathers official database adapters. For specifics on how to use the database adapters see the [database adapters common API](#).

.find(params)

`find(params) -> Promise` - retrieves a list of all resources from the service. Provider parameters will be passed as `params.query`.

```
app.use('/messages', {
  find(params) {
    return Promise.resolve([
      {
        id: 1,
        text: 'Message 1'
      },
      {
        id: 2,
        text: 'Message 2'
      }
    ]);
  }
});
```

Note: `find` does not have to return an array it can also return an object. The database adapters already do this for [pagination](#).

.get(id, params)

`get(id, params) -> Promise` - retrieves a single resource with the given `id` from the service.

```
app.use('/messages', {
  get(id, params) {
    return Promise.resolve({
      id,
      text: `You have to do ${id}!`
    });
  }
});
```

.create(data, params)

`create(data, params) -> Promise` - creates a new resource with `data`. The method should return a Promise with the newly created data. `data` may also be an array.

```
app.use('/messages', {
  messages: [],

  create(data, params) {
    this.messages.push(data);

    return Promise.resolve(data);
  }
});
```

Important: A successful `create` method call emits the `created` service event.

.update(id, data, params)

`update(id, data, params) -> Promise` - replaces the resource identified by `id` with `data`. The method should return a Promise with the complete updated resource data. `id` can also be `null` when updating multiple records with `params.query` containing the query criteria.

Important: A successful `update` method call emits the `updated` service event.

.patch(id, data, params)

`patch(id, data, params) -> Promise` - merges the existing data of the resource identified by `id` with the new `data`. `id` can also be `null` indicating that multiple resources should be patched with `params.query` containing the query criteria.

The method should return with the complete updated resource data. Implement `patch` additionally (or instead of) `update` if you want to separate between partial and full updates and support the `PATCH` HTTP method.

Important: A successful `patch` method call emits the `patched` service event.

.remove(id, params)

`remove(id, params) -> Promise` - removes the resource with `id`. The method should return a Promise with the removed resource. `id` can also be `null` indicating to delete multiple resources with `params.query` containing the query criteria.

Important: A successful `remove` method call emits the `removed` service event.

.setup(app, path)

`setup(app, path) -> Promise` is a special method that initializes the service, passing an instance of the Feathers application and the path it has been registered on.

For services registered before `app.listen` is invoked, the `setup` function of each registered service is called upon invoking `app.listen`. For services registered after `app.listen` is invoked, `setup` is called automatically by Feathers when a service is registered.

`setup` is a great place to initialize your service with any special configuration or if connecting services that are very tightly coupled (see below), as opposed to using [hooks](#).

```
// app.js
'use strict';

const feathers = require('feathers');
const rest = require('feathers-rest');

class MessageService {
  get(id, params) {
    return Promise.resolve({
      id,
      read: false,
      text: `Feathers is great!`,
```

```

        createdAt: new Date.getTime()
    });
}
}

class MyService {
  setup(app) {
    this.app = app;
  }

  get(name, params) {
    const messages = this.app.service('messages');

    return messages.get(1)
      .then(message => {
        return { name, message };
      });
  }
}

const app = feathers()
  .configure(rest())
  .use('/messages', new MessageService())
  .use('/my-service', new MyService())

app.listen(3030);

```

Feathers functionality

When registering a service, Feathers (or its plugins) can also add its own methods to a service. Most notably, every service will automatically become an instance of a [NodeJS EventEmitter](#).

.hooks(hooks)

Register [hooks](#) for this service.

.filter(filters)

Register a set of [event filters](#) to filter Feathers real-time events to specific clients.

.on(eventname, listener)

Provided by the core [NodeJS EventEmitter .on](#). Registers a `listener` method (`function(data) {}`) for the given `eventname`.

Important: For more information about service event see the [Events chapter](#).

.emit(eventname, data)

Provided by the core [NodeJS EventEmitter .emit](#). Emits the event `eventname` to all event listeners.

Important: For more information about service event see the [Events chapter](#).

.removeListener(eventname, [listener])

Provided by the core [NodeJS EventEmitter .removeListener](#). Removes all or the given listener for `eventname`.

Important: For more information about service event see the [Events chapter](#).

Hooks



```
$ npm install feathers-hooks --save
```

Hooks are pluggable middleware functions that can be registered **before**, **after** or on **errors** of a [service method](#). You can register a single hook function or create a chain of them to create complex work-flows. Most of the time multiple hooks are registered so the examples show the "hook chain" array style registration.

A hook is **transport independent**, which means it does not matter if it has been called through HTTP(S) (REST), Socket.io, Primus or any other transport Feathers may support in the future. They are also service agnostic, meaning they can be used with **any** service regardless of whether they have a model or not.

Hooks are commonly used to handle things like validation, logging, populating related entities, sending notifications and more. This pattern keeps your application logic flexible, composable, and much easier to trace through and debug. For more information about the design patterns behind hooks see [this blog post](#).

The following example adds a `createdAt` and `updatedAt` property before sending the data to the database.

```
const feathers = require('feathers');
const hooks = require('feathers-hooks');

const app = feathers();

app.configure(hooks());

app.service('messages').hooks({
  before: [
    create(hook) {
      hook.data.createdAt = new Date();
    },
    update(hook) {
      hook.data.updatedAt = new Date();
    },
    patch(hook) {
      hook.data.updatedAt = new Date();
    }
  ]
});
```

Hook objects

The `hook` object is passed to a hook function and contains information about the service method call. Hook objects have **read only** properties that should not be modified and **writable** properties that can be changed for subsequent hooks.

- **Read Only:**

- `app` - The [app object](#) (used to e.g. retrieve other services)
- `service` - The service this hook currently runs on
- `path` - The path (name) of the service
- `method` - The service method name
- `type` - The hook type (`before`, `after` or `error`)

- **Writable:**

- `params` - The service method parameters (including `params.query`)
- `id` - The id (for `get`, `remove`, `update` and `patch`)
- `data` - The request data (for `create`, `update` and `patch`)
- `error` - The error that was thrown (only in `error` hooks)
- `result` - The result of the successful method call (only in `after` hooks).

Pro Tip: `hook.result` Can also be set in a `before` hook which will skip the service method call (but run all other hooks).

Pro Tip: `hook.id` can also be `null` for `update`, `patch` and `remove`. See the [service methods](#) for more information.

Pro Tip: The `hook` object is the same throughout a service method call so it is possible to add properties and use them in other hooks at a later time.

Hook functions

A hook function (or just `hook`) takes a [hook object](#) as the parameter (`function(hook) {}` or `hook => {}`) and can

- return nothing (`undefined`)
- return the `hook` object
- `throw` an error
- for asynchronous operations return a [Promise](#) that
 - resolves with a `hook` object
 - resolves with `undefined`
 - rejects with an error

When an error is thrown (or the promise is rejected), all subsequent hooks - and the service method call if it didn't run already - will be skipped and only the error hooks will run.

The following example throws an error when the text for creating a new message is empty. You can also create very similar hooks to use your Node validation library of choice.

```
app.service('messages').hooks({
  before: {
    create: [
      function(hook) {
        if(hook.data.text.trim() === '') {
          throw new Error('Message text can not be empty');
        }
      }
    ]
  }
});
```

Asynchronous hooks

When a Promise is returned the hook will wait until it resolves or rejects before continuing.

Important: As stated in the [hook functions](#) section the promise has to either resolve with the `hook` object (usually done with `.then(() => hook)` at the end of the promise chain) or with `undefined`.

The following example shows an asynchronous hook that uses another service to retrieve and populate the messages `user` when getting a single message.

```
app.service('messages').hooks({
  after: [
    get: [
      function(hook) {
        const userId = hook.result.userId;

        // hook.app.service('users').get returns a Promise already
        return hook.app.service('users').get(userId).then(user => {
          // Update the result (the message)
          hook.result.user = user;

          // Returning will resolve the promise with the `hook` object
          return hook;
        });
      }
    ]
  ]
});
```

When the asynchronous operation is using a `callback` instead of returning a promise you have to create and return a new Promise (`new Promise((resolve, reject) => {})`).

The following example reads a JSON file with `fs.readFile` and adds it to the message:

```
app.service('messages').hooks({
  after: [
    get: [
      function(hook) {
        return new Promise((resolve, reject) => {
          require('fs').readFile('./myfile.json', (error, data) => {
            // Check if the callback got an error, if so reject the promise and return
            if(error) {
              return reject(error);
            }

            hook.result.myFile = JSON.parse(data.toString());

            // Resolve the promise with the `hook` object
            resolve(hook);
          });
        });
      ]
    ]
  ]
});
```

Pro Tip: Tools like [Bluebird](#) make converting between callbacks and promises easier.

Important: Most Feathers service calls and newer Node packages already return Promises. They can be returned and chained directly. There is no need to instantiate your own `new Promise` instance in those cases.

Registering hooks

Hook functions are registered on a service through the `app.service(<servicename>).hooks(hooks)` method. There are several options for what can be passed as `hooks`:

```
// The standard all at once way (also used by the generator)
// an array of functions per service method name (and for `all` methods)
app.service('servicename').hooks({
  before: {
    all: [
      // Use normal functions
```

```

        function(hook) { console.log('before all hook ran'); }
    ],
    find: [
        // Use ES6 arrow functions
        hook => console.log('before find hook 1 ran'),
        hook => console.log('before find hook 2 ran')
    ],
    get: [ /* other hook functions here */ ],
    create: [],
    update: [],
    patch: [],
    remove: []
},
after: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
},
error: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
}
});

// Register a single hook before, after and on error for all methods
app.service('servicename').hooks({
    before(hook) {
        console.log('before all hook ran');
    },
    after(hook) {
        console.log('after all hook ran');
    },
    error(hook) {
        console.log('error all hook ran');
    }
});

```

Pro Tip: When using the full object, `all` is a special keyword meaning this hook will run for all methods. `all` hooks will be registered before other method specific hooks.

Pro Tip: `app.service(<servicename>).hooks(hooks)` can be called multiple times and the hooks will be registered in that order. Normally all hooks should be registered at once however to see at a glance what the service is going to do.

Application hooks

To add hooks to every service `app.hooks(hooks)` can be used. Application hooks are [registered in the same format as service hooks](#) and also work exactly the same. Note when application hooks will be executed however:

- `before` application hooks will always run *before* all service `before` hooks
- `after` application hooks will always run *after* all service `after` hooks
- `error` application hooks will always run *after* all service `error` hooks

Here is an example for a very useful application hook that logs every service method error with the service and method name as well as the error stack.

```
app.hooks({
  error(hook) {
    console.error(`Error in '${hook.path}' service method '${hook.method}', ${hook.error.stack}`);
  }
});
```

Common Hooks

[Star 42](#) [npm v3.7.2](#) [changelog .md](#)

```
$ npm install feathers-hooks-common --save
```

`feathers-hooks-common` is a collection of common [hooks](#) and utilities.

[Authentication hooks](#) are documented separately.

Note: Many hooks are just a few lines of code to implement from scratch. If you can't find a hook here but are unsure how to implement it or have an idea for a generally useful hook create a new issue [here](#).

client

`client(... whitelist)` [source](#)

A hook for passing params from the client to the server.

- Used as a `before` hook.

ProTip Use the `paramsFromClient` hook instead. It does exactly the same thing as `client` but is less likely to be deprecated.

Only the `hook.params.query` object is transferred to the server from a Feathers client, for security among other reasons. However if you can include a `hook.params.query.$client` object, e.g.

```
service.find({
  query: {
    dept: 'a',
    $client: {
      populate: 'po-1',
      serialize: 'po-mgr'
    }
  }
});
```

the `client` hook will move that data to `hook.params` on the server.

```
service.before({ all: [ client('populate', 'serialize', 'otherProp'), myHook ]});
// myHook's hook.params will be
// { query: { dept: 'a' }, populate: 'po-1', serialize: 'po-mgr' } }
```

Options:

- `whitelist` (*optional*) Names of the potential props to transfer from `query.client`. Other props are ignored. This is a security feature.

ProTip You can use the same technique for service calls made on the server.

See `Util: paramsForServer` and `paramsFromClient`.

combine

combine(. . . hookFuncs) [source](#)

Sequentially execute multiple hooks within a custom hook function.

```
function (hook) { // an arrow func cannot be used because we need 'this'  
// ...  
hooks.combine(hook1, hook2, hook3).call(this, hook)  
.then(hook => {});  
}
```

Options:

- `hooks` (*optional*) - The hooks to run.

ProTip: `combine` is primarily intended to be used within your custom hooks, not when [registering hooks](#). Its more convenient to use the following when registering hooks:

```
const workflow = [hook1(), hook2(), ...];  
app.service(...).hooks({  
  before: {  
    update: [...workflow],  
    patch: [...workflow],  
  },  
});
```

debug

debug(label) [source](#)

Display current info about the hook to console.

- Used as a `before` or `after` hook.

```
const { debug } = require('feathers-hooks-common');  
  
debug('step 1')  
// * step 1  
// type: before, method: create  
// data: { name: 'Joe Doe' }  
// query: { sex: 'm' }  
// result: { assigned: true }
```

Options:

- `label` (*optional*) - Label to identify the debug listing.

dePopulate

dePopulate() [source](#)

Removes joined and [computed](#) properties, as well any profile information. Populated and serialized items may, after `dePopulate`, be used in `service.patch(id, items)` calls.

- Used as a `before` or `after` hook on any service method.
- Supports multiple result items, including paginated `find`.
- Supports an array of keys in `field`.

See also `populate`, `serialize`.

disallow

disallow(. . . providers) [source](#)

Disallows access to a service method completely or for specific providers. All providers (REST, Socket.io and Primus) set the `hook.params.provider` property, and `disallow` checks this.

- Used as a `before` hook.

```
app.service('users').before({
  // Users can not be created by external access
  create: hooks.disallow('external'),
  // A user can not be deleted through the REST provider
  remove: hooks.disallow('rest'),
  // disallow calling `update` completely (e.g. to allow only `patch`)
  update: hooks.disallow(),
  // disallow the remove hook if the user is not an admin
  remove: hooks.when(hook => !hook.params.user.isAdmin, hooks.disallow())
});
```

ProTip Service methods that are not implemented do not need to be disallowed.

Options:

- `providers` (*optional*, default: disallows everything) - The transports that you want to disallow this service method for. Options are:
 - `socketio` - will disallow the method for the Socket.IO provider
 - `primus` - will disallow the method for the Primus provider
 - `rest` - will disallow the method for the REST provider
 - `external` - will disallow access from all providers other than the server.
 - `server` - will disallow access for the server

disableMultiItemChange

disableMultiItemChange() [source](#)

Disables update, patch and remove methods from using null as an id, e.g. `remove(null)`. A null id affects all the items in the DB, so accidentally using it may have undesirable results.

- Used as a `before` hook.

```
app.service('users').before({
  update: hooks.disableMultiItemChange(),
});
```

discard

discard(. . . fieldNames) [source](#)

Delete the given fields either from the data submitted or from the result. If the data is an array or a paginated `find` result the hook will Delete the field(s) for every item.

- Used as a `before` hook for `create`, `update` or `patch`.
- Used as an `after` hook.
- Field names support dot notation e.g. `name.address.city`.
- Supports multiple data items, including paginated `find`.

```
const { discard } = require('feathers-hooks-common');

// Delete the hashed `password` and `salt` field after all method calls
app.service('users').after(discard('password', 'salt'));

// Delete _id for `create`, `update` and `patch`
app.service('users').before({
  create: discard('_id', 'password'),
  update: discard('_id'),
  patch: discard('_id')
})
```

ProTip: This hook will always delete the fields, unlike the `remove` hook which only deletes the fields if the service call was made by a client.

ProTip: You can replace `remove('name')` with `iff(isProvider('external'), discard('name'))`. The latter does not contain any hidden "magic".

Options:

- `fieldNames` (*required*) - One or more fields you want to remove from the object(s).

See also `remove`.

else

iff(...).else(...hookFuncs) [source](#)

`iff().else()` is similar to `iff` and `iffElse`. Its syntax is more suitable for writing nested conditional hooks. If the predicate in the `iff()` is falsey, run the hooks in `else()` sequentially.

- Used as a `before` or `after` hook.
- Hooks to run may be sync, Promises or callbacks.
- `feathers-hooks` catches any errors thrown in the predicate or hook.

```
service.before({
  create:
    hooks.iff(isProvider('server'),
      hookA,
      hooks.iff(isProvider('rest'), hook1, hook2, hook3)
        .else(hook4, hook5),
      hookB
    )
    .else(
      hooks.iff(hook => hook.path === 'users', hook6, hook7)
    )
});
```

OR:

```
service.before({
  create:
```

```
hooks.ifff(isServer, [
  hookA,
  hooks.ifff(isProvider('rest'), [hook1, hook2, hook3])
    .else([hook4, hook5]),
  hookB
])
.else([
  hooks.ifff(hook => hook.path === 'users', [hook6, hook7])
])
});
```

Options:

- `hookFuncs` (*optional*) - Zero or more hook functions. They may include other conditional hooks. Or you can use an array of hook functions as the second parameter.

See also `iff`, `iffElse`, `when`, `unless`, `isNot`, `isProvider`.

This The predicate and hook functions in the `if`, `else` and `iffElse` hooks will not be called with `this` set to the service. Use `hook.service` instead.

every

`every(... hookFuncs)` [source](#)

Run hook functions in parallel. Return `true` if every hook function returned a truthy value.

- Used as a predicate function with conditional hooks.
- The current `hook` is passed to all the hook functions, and they are run in parallel.
- Hooks to run may be sync or Promises only.
- `feathers-hooks` catches any errors thrown in the predicate.

```
service.before({
  create: hooks.ever(hooks.every(hook1, hook2, ...), hookA, hookB, ...)
});
```

```
hooks.every(hook1, hook2, ...).call(this, currentHook)
.then(bool => { ... });
```

Options:

- `hookFuncs` (*required*) Functions which take the current hook as a param and return a boolean result.

See also `some`.

iff

`iff(predicate: boolean|Promise|function, ...hookFuncs: HookFunc[]): HookFunc` [source](#)

Resolve the predicate to a boolean. Run the hooks sequentially if the result is truthy.

- Used as a `before` or `after` hook.
- Predicate may be a sync or async function.
- Hooks to run may be sync, Promises or callbacks.

- `feathers-hooks` catches any errors thrown in the predicate or hook.

```
const { iff, populate } = require('feathers-hooks-common');
const isNotAdmin = adminRole => hook => hook.params.user.roles.indexOf(adminRole || 'admin') === -1;

app.service('workOrders').after({
  // async predicate and hook
  create: iff(
    () => new Promise((resolve, reject) => { ... }),
    populate('user', { field: 'authorisedById', service: 'users' })
  )
});

app.service('workOrders').after({
  // sync predicate and hook
  find: [ iff(isNotAdmin(), hooks.remove('budget')) ]
});
```

or with the array syntax:

```
app.service('workOrders').after({
  find: [ iff(isNotAdmin(), [hooks.remove('budget'), hooks.remove('password')]) ]
});
```

Options:

- `predicate` (*required*) - Determines if hookFuncs should be run or not. If a function, `predicate` is called with the hook as its param. It returns either a boolean or a Promise that evaluates to a boolean
- `hookFuncs` (*optional*) - Zero or more hook functions. They may include other conditional hooks. Or you can use an array of hook functions as the second parameter.

See also `iffElse`, `else`, `when`, `unless`, `isNot`, `isProvider`.

This The predicate and hook functions in the `if`, `else` and `iffElse` hooks will not be called with `this` set to the service. Use `hook.service` instead.

iffElse

iffElse(predicate, trueHooks, falseHooks) [source](#)

Resolve the predicate to a boolean. Run the first set of hooks sequentially if the result is truthy, the second otherwise.

- Used as a `before` or `after` hook.
- Predicate may be a sync or async function.
- Hooks to run may be sync, Promises or callbacks.
- `feathers-hooks` catches any errors thrown in the predicate or hook.

```
const { iffElse, populate, serialize } = require('feathers-hooks-common');
app.service('purchaseOrders').after({
  create: iffElse(() => { ... },
    [populate(poAccting), serialize( ... )],
    [populate(poReceiving), serialize( ... )]
  )
});
```

Options:

- `predicate` (*required*) - Determines if hookFuncs should be run or not. If a function, `predicate` is called with the hook as its param. It returns either a boolean or a Promise that evaluates to a boolean
- `trueHooks` (*optional*) - Zero or more hook functions run when `predicate` is truthy.
- `falseHooks` (*optional*) - Zero or more hook functions run when `predicate` is false.

See also `iff`, `else`, `when`, `unless`, `isNot`, `isProvider`.

This The predicate and hook functions in the `if`, `else` and `iffElse` hooks will not be called with `this` set to the service. Use `hook.service` instead.

isNot

`isNot(predicate)` [source](#)

Negate the `predicate`.

- Used as a predicate with conditional hooks.
- Predicate may be a sync or async function.
- `feathers-hooks` catches any errors thrown in the predicate.

```
import hooks, { iff, isNot, isProvider } from 'feathers-hooks-common';
const isRequestor = () => hook => new Promise(resolve, reject) => ... );

app.service('workOrders').after({
  iff(isNot(isRequestor()), hooks.remove( ... ))
});
```

Options:

- `predicate` (*required*) - A function which returns either a boolean or a Promise that resolves to a boolean.

See also `iff`, `iffElse`, `else`, `when`, `unless`, `isProvider`.

isProvider

`isProvider(provider)` [source](#)

Check which transport called the service method. All providers ([REST](#), [Socket.io](#) and [Primus](#)) set the `params.provider` property which is what `isProvider` checks for.

- Used as a predicate function with conditional hooks.

```
import { iff, isProvider, remove } from 'feathers-hooks-common';

app.service('users').after({
  iff(isProvider('external'), remove( ... ))
});
```

Options:

- `provider` (*required*) - The transport that you want this hook to run for. Options are:
 - `server` - Run the hook if the server called the service method.
 - `external` - Run the hook if any transport other than the server called the service method.
 - `socketio` - Run the hook if the Socket.IO provider called the service method.
 - `primus` - If the Primus provider.

- o `rest` - If the REST provider.
- `providers` (*optional*) - Other transports that you want this hook to run for.

See also `iff`, `iffElse`, `else`, `when`, `unless`, `isNot`, `isProvider`.

lowerCase

lowerCase(... `fieldNames`) [source](#)

Lower cases the given fields either in the data submitted or in the result. If the data is an array or a paginated `find` result the hook will lowercase the field(s) for every item.

- Used as a `before` hook for `create`, `update` or `patch`.
- Used as an `after` hook.
- Field names support dot notation.
- Supports multiple data items, including paginated `find`.

```
const { lowerCase } = require('feathers-hooks-common');

// lowercase the `email` and `password` field before a user is created
app.service('users').before({
  create: lowerCase('email', 'username')
});
```

Options:

- `fieldNames` (*required*) - One or more fields that you want to lowercase from the retrieved object(s).

See also `upperCase`.

paramsFromClient

paramsFromClient(... `whitelist`) [source](#)

A hook, on the server, for passing `params` from the client to the server.

- Used as a `before` hook.
- Companion to the client utility function `paramsForServer`.

By default, only the `hook.params.query` object is transferred to the server from a Feathers client, for security among other reasons. However you can explicitly transfer other `params` props with the client utility function `paramsForServer` in conjunction with the hook function `paramsFromClient` on the server.

```
// client
import { paramsForServer } from 'feathers-hooks-common';
service.patch(null, data, paramsForServer({
  query: { dept: 'a' }, populate: 'po-1', serialize: 'po-mgr'
}));

// server
const { paramsFromClient } = require('feathers-hooks-common');
service.before({ all: [
  paramsFromClient('populate', 'serialize', 'otherProp'), myHook
]});

// hook.params will now be
// { query: { dept: 'a' }, populate: 'po-1', serialize: 'po-mgr' }
```

Options:

- `whitelist` (*optional*) Names of the permitted props; other props are ignored. This is a security feature.

ProTip You can use the same technique for service calls made on the server.

See `util: paramsForServer`.

pluck

`pluck(... fieldNames) source`

Discard all other fields except for the provided fields either from the data submitted or from the result. If the data is an array or a paginated `find` result the hook will remove the field(s) for every item.

- Used as a `before` hook for `create`, `update` or `patch`.
- Used as an `after` hook.
- Field names support dot notation.
- Supports multiple data items, including paginated `find`.

```
const { pluck } = require('feathers-hooks-common');

// Only retain the hashed `password` and `salt` field after all method calls
app.service('users').after(pluck('password', 'salt'));

// Only keep the _id for `create`, `update` and `patch`
app.service('users').before({
  create: pluck('_id'),
  update: pluck('_id'),
  patch: pluck('_id')
})
```

ProTip: This hook will only fire when `params.provider` has a value, i.e. when it is an external request over REST or Sockets.

Options:

- `fieldNames` (*required*) - One or more fields that you want to retain from the object(s).

All other fields will be discarded.

pluckQuery

`pluckQuery(... fieldNames) source`

Discard all other fields except for the given fields from the query params.

- Used as a `before` hook.
- Field names support dot notation.
- Supports multiple data items, including paginated `find`.

```
const { pluckQuery } = require('feathers-hooks-common');

// Discard all other fields except for _id from the query
// for all service methods
app.service('users').before({
  all: pluckQuery('_id')
});
```

ProTip: This hook will only fire when `params.provider` has a value, i.e. when it is an external request over REST or Sockets.

Options:

- `fieldNames` (*optional*) - The fields that you want to retain from the query object. All other fields will be discarded.

populate

populate(options: Object): HookFunc [source](#)

Populates items *recursively* to any depth. Supports 1:1, 1:n and n:1 relationships.

- Used as a **before** or **after** hook on any service method.
- Supports multiple result items, including paginated `find`.
- Permissions control what a user may see.
- Provides performance profile information.
- Backward compatible with the old FeathersJS `populate` hook.

Examples

- 1:1 relationship

```
// users like { _id: '111', name: 'John', roleId: '555' }
// roles like { _id: '555', permissions: ['foo', 'bar'] }
import { populate } from 'feathers-hooks-common';

const userRoleSchema = {
  include: {
    service: 'roles',
    nameAs: 'role',
    parentField: 'roleId',
    childField: '_id'
  }
};

app.service('users').hooks({
  after: {
    all: populate({ schema: userRoleSchema })
  }
});

// result like
// { _id: '111', name: 'John', roleId: '555',
//   role: { _id: '555', permissions: ['foo', 'bar'] } }
```

- 1:n relationship

```
// users like { _id: '111', name: 'John', roleIds: ['555', '666'] }
// roles like { _id: '555', permissions: ['foo', 'bar'] }
const userRolesSchema = {
  include: {
    service: 'roles',
    nameAs: 'roles',
    parentField: 'roleIds',
    childField: '_id'
  }
};

usersService.hooks({
```

```

    after: {
      all: populate({ schema: userRolesSchema })
    }
});

// result like
// { _id: '111', name: 'John', roleIds: ['555', '666'], roles: [
//   { _id: '555', permissions: ['foo', 'bar'] }
//   { _id: '666', permissions: ['fiz', 'buz'] }
// ]}

```

- n:1 relationship

```

// posts like { _id: '111', body: '...' }
// comments like { _id: '555', text: '...', postId: '111' }
const postCommentsSchema = {
  include: {
    service: 'comments',
    nameAs: 'comments',
    parentField: '_id',
    childField: 'postId'
  }
};

postService.hooks({
  after: {
    all: populate({ schema: postCommentsSchema })
  }
});

// result like
// { _id: '111', body: '...', comments: [
//   { _id: '555', text: '...', postId: '111' }
//   { _id: '666', text: '...', postId: '111' }
// ]}

```

- Multiple and recursive includes

```

const schema = {
  service: '...',
  permissions: '...',
  include: [
    {
      service: 'users',
      nameAs: 'authorItem',
      parentField: 'author',
      childField: 'id',
      include: [ ... ],
    },
    {
      service: 'comments',
      parentField: 'id',
      childField: 'postId',
      query: {
        $limit: 5,
        $select: ['title', 'content', 'postId'],
        $sort: {createdAt: -1}
      },
      select: (hook, parent, depth) => ({ $limit: 6 }),
      asArray: true,
      provider: undefined,
    },
    {
      service: 'users',
      permissions: '...',
      nameAs: 'readers',
    }
  ]
};

```

```

        parentField: 'readers',
        childField: 'id'
    }
],
};

module.exports.after = {
    all: populate({ schema, checkPermissions, profile: true })
};

```

- Flexible relationship, similar to the n:1 relationship example above

```

// posts like { _id: '111', body: '...' }
// comments like { _id: '555', text: '...', postId: '111' }
const postCommentsSchema = {
    include: {
        service: 'comments',
        nameAs: 'comments',
        select: (hook, parentItem) => ({ postId: parentItem._id }),
    }
};

postService.hooks({
    after: {
        all: populate({ schema: postCommentsSchema })
    }
});

// result like
// { _id: '111', body: '...', comments: [
//     { _id: '555', text: '...', postId: '111' },
//     { _id: '666', text: '...', postId: '111' }
// ]}

```

Options

- `schema` (*required*, object or function) How to populate the items. [Details are below.](#)
 - Function signature `(hook: Hook, options: Object): Object`
 - `hook` The hook.
 - `options` The options passed to the populate hook.
- `checkPermissions` [optional, default () => true] Function to check if the user is allowed to perform this populate, or include this type of item. Called whenever a `permissions` property is found.
 - Function signature `(hook: Hook, service: string, permissions: any, depth: number): boolean`
 - `hook` The hook.
 - `service` The name of the service being included, e.g. users, messages.
 - `permissions` The value of the permissions property.
 - `depth` How deep the include is in the schema. Top of schema is 0.
 - Return `true` to allow the include.
- `profile` [optional, default false] If `true`, the populated result is to contain a performance profile. Must be `true`, `true` is insufficient.

Schema

The data currently in the hook will be populated according to the schema. The schema starts with:

```

const schema = {
    service: '...',
    permissions: '...',
    include: [ ... ]
}

```

```
};
```

- `service` (*optional*) The name of the service this schema is to be used with. This can be used to prevent a schema designed to populate 'blog' items from being incorrectly used with `comment` items.
- `permissions` (*optional*, any type of value) Who is allowed to perform this populate. See `checkPermissions` above.
- `include` (*optional*) Which services to join to the data.

Include

The `include` array has an element for each service to join. They each may have:

```
{
  service: 'comments',
  nameAs: 'commentItems',
  permissions: '...',
  parentField: 'id',
  childField: 'postId',
  query: {
    $limit: 5,
    $select: ['title', 'content', 'postId'],
    $sort: {createdAt: -1}
  },
  select: (hook, parent, depth) => ({ $limit: 6 }),
  asArray: true,
  paginate: false,
  provider: undefined,
  useInnerPopulate: false,
  include: [ ... ]
}
```

ProTip Instead of setting `include` to a 1-element array, you can set it to the include object itself, e.g. `include: { ... }`.

- `service` [*required*, string] The name of the service providing the items.
- `nameAs` [*optional*, string, default is `service`] Where to place the items from the join. Dot notation is allowed.
- `permissions` [*optional*, any type of value] Who is allowed to perform this join. See `checkPermissions` above.
- `parentField` [*required if neither query nor select*, string] The name of the field in the parent item for the `relation`. Dot notation is allowed.
- `childField` [*required if neither query nor select*, string] The name of the field in the child item for the `relation`. Dot notation is allowed and will result in a query like `{ 'name.first': 'John' }` which is not suitable for all DBs. You may use `query` or `select` to create a query suitable for your DB.
- `query` [*optional*, object] An object to inject into the query in `service.find({ query: { ... } })`.
- `select` [*optional*, function] A function whose result is injected into the query.
 - Function signature `(hook: Hook, parentItem: Object, depth: number): Object`
 - `hook` The hook.
 - `parentItem` The parent item to which we are joining.
 - `depth` How deep the include is in the schema. Top of schema is 0.
- `asArray` [*optional*, boolean, default false] Force a single joined item to be stored as an array.
- `paginate` [*optional*, boolean or number, default false] Controls pagination for this service.
 - `false` No pagination. The default.
 - `true` Use the configuration provided when the service was configured/
 - A number. The maximum number of items to include.
- `provider` [*optional*] `find` calls are made to obtain the items to be joined. These, by default, are initialized to look like they were made by the same provider as that getting the base record. So when populating the result of a call made via `socketio`, all the join calls will look like they were made via `socketio`. Alternative you can set `provider: undefined` and the calls for that join will look like they were made by the server. The hooks on the service may behave differently in different situations.

- `useInnerPopulate` [optional] Populate, when including records from a child service, ignores any populate hooks defined for that child service. The `useInnerPopulate` option will run those populate hooks. This allows the populate for a base record to include child records containing their own immediate child records, without the populate for the base record knowing what those grandchildren populates are.
- `include` [optional] The new items may themselves include other items. The includes are recursive.

Populate forms the query `[childField]: parentItem[parentField]` when the parent value is not an array. This will include all child items having that value.

Populate forms the query `[childField]: { $in: parentItem[parentField] }` when the parent value is an array. This will include all child items having any of those values.

A populate hook for, say, `posts` may include items from `users`. Should the `users` hooks also include a populate, that `users` populate hook will not be run for includes arising from `posts`.

ProTip The populate interface only allows you to directly manipulate `hook.params.query`. You can manipulate the rest of `hook.params` by using the `client` hook, along with something like `query: { ... }, $client: { paramsProp1: ..., paramsProp2: ... } }`.

Added properties

Some additional properties are added to populated items. The result may look like:

```
{
  ...
  _include: [ 'post' ],
  _elapsed: { post: 487947, total: 527118 },
  post:
  {
    ...
    _include: [ 'authorItem', 'commentsInfo', 'readersInfo' ],
    _elapsed: { authorItem: 321973, commentsInfo: 469375, readersInfo: 479874, total: 487947 },
    _computed: [ 'averageStars', 'views' ],
    authorItem: { ... },
    commentsInfo: [ { ... }, { ... } ],
    readersInfo: [ { ... }, { ... } ]
  }
}
```

- `_include` The property names containing joined items.
- `_elapsed` The elapsed time in nano-seconds (where 1,000,000 ns === 1 ms) taken to perform each include, as well as the total taken for them all. This delay is mostly attributed to your DB.
- `_computed` The property names containing values computed by the `serialize` hook.

The `depopulate` hook uses these fields to remove all joined and computed values. This allows you to then `service.patch()` the item in the hook.

Joining without using related fields

Populate can join child records to a parent record using the related columns `parentField` and `childField`. However populate's `query` and `select` options may be used to relate the records without needing to use the related columns. This is a more flexible, non-SQL-like way of relating records. It easily supports dynamic, run-time schemas since the `select` option may be a function.

Populate examples

Selecting schema based on UI needs

Consider a Purchase Order item. An Accounting oriented UI will likely want to populate the PO with Invoice items. A Receiving oriented UI will likely want to populate with Receiving Slips.

Using a function for `schema` allows you to select an appropriate schema based on the need. The following example shows how the client can ask for the type of schema it needs.

```
// on client
import { paramsForServer } from 'feathers-hooks-common';
purchaseOrders.get(id, paramsForServer({ schema: 'po-acct' })); // pass schema name to server
// or
purchaseOrders.get(id, paramsForServer({ schema: 'po-rec' }));
```

```
// on server
import { paramsFromClient } from 'feathers-hooks-common';
const poSchemas = {
  'po-acct': /* populate schema for Accounting oriented PO e.g. { include: ... } */,
  'po-rec': /* populate schema for Receiving oriented PO */
};

purchaseOrders.before({
  all: paramsFromClient('schema')
});

purchaseOrders.after({
  all: populate({ schema: hook => poSchemas[hook.params.schema] })
});
```

Using permissions

For a simplistic example, assume `hook.params.users.permissions` is an array of the service names the user may use, e.g. `['invoices', 'billings']`. These can be used to control which types of items the user can see.

The following `populate` will only be performed for users whose `user.permissions` contains `'invoices'`.

```
const schema = {
  include: [
    {
      service: 'invoices',
      permissions: 'invoices',
      ...
    }
  ]
};

purchaseOrders.after({
  all: populate(schema, (hook, service, permissions) => hook.params.user.permissions.includes(service))
});
```

See also `dePopulate`, `serialize`.

preventChanges

`preventChanges(... fieldNames)` [source](#)

Prevents the specified fields from being patched.

- Used as a `before` hook for `patch`.
- Field names support dot notation e.g. `name.address.city`.

```
const { preventChanges } = require('feathers-hooks-common');

app.service('users').before({
```

```
patch: preventChanges('security.badge')
})
```

Options:

- `fieldNames` (*required*) - One or more fields which may not be patched.

Consider using `validateSchema` if you would rather specify which fields are allowed to change.

remove

`remove(... fieldNames)` [source](#)

Remove the given fields either from the data submitted or from the result. If the data is an array or a paginated `find` result the hook will remove the field(s) for every item.

- Used as a `before` hook for `create`, `update` or `patch`.
- Used as an `after` hook.
- Field names support dot notation e.g. `name.address.city`.
- Supports multiple data items, including paginated `find`.

```
const { remove } = require('feathers-hooks-common');

// Remove the hashed `password` and `salt` field after all method calls
app.service('users').after(remove('password', 'salt'));

// Remove _id for `create`, `update` and `patch`
app.service('users').before({
  create: remove('_id', 'password'),
  update: remove('_id'),
  patch: remove('_id')
})
```

ProTip: This hook will only fire when `params.provider` has a value, i.e. when it is an external request over REST or Sockets.

Options:

- `fieldNames` (*required*) - One or more fields you want to remove from the object(s).

See also `discard`.

removeQuery

`removeQuery(... fieldNames)` [source](#)

Remove the given fields from the query params.

- Used as a `before` hook.
- Field names support dot notation
- Supports multiple data items, including paginated `find`.

```
const { removeQuery } = require('feathers-hooks-common');

// Remove _id from the query for all service methods
app.service('users').before({
  all: removeQuery('_id')
});
```

ProTip: This hook will only fire when `params.provider` has a value, i.e. when it is an external request over REST or Sockets.

Options:

- `fieldNames` (*optional*) - The fields that you want to remove from the query object.

serialize

`serialize(schema: Object|Function): HookFunc source`

Remove selected information from populated items. Add new computed information. Intended for use with the `populate` hook.

```
const schema = {
  only: 'updatedAt',
  computed: {
    commentsCount: (recommendation, hook) => recommendation.post.commentsInfo.length,
  },
  post: {
    exclude: ['id', 'createdAt', 'author', 'readers'],
    authorItem: {
      exclude: ['id', 'password', 'age'],
      computed: {
        isUnder18: (authorItem, hook) => authorItem.age < 18,
      },
    },
    readersInfo: {
      exclude: 'id',
    },
    commentsInfo: {
      only: ['title', 'content'],
      exclude: 'content',
    },
  },
};
purchaseOrders.after({
  all: [ populate( ... ), serialize(schema) ]
});
```

Options

- `schema` [*required*, object or function] How to serialize the items.
 - Function signature `(hook: Hook): Object`
 - `hook` The hook.

The schema reflects the structure of the populated items. The base items for the example above have included `post` items, which themselves have included `authorItem`, `readersInfo` and `commentsInfo` items.

The schema for each set of items may have

- `only` [*optional*, string or array of strings] The names of the fields to keep in each item. The names for included sets of items plus `_include` and `_elapsed` are not removed by `only`.
- `exclude` [*optional*, string or array of strings] The names of fields to drop in each item. You may drop, at your own risk, names of included sets of items, `_include` and `_elapsed`.
- `computed` [*optional*, object with functions] The new names you want added and how to compute their values.
 - Object is like `{ name: func, ... }`
 - `name` The name of the field to add to the items.

- o `func` Function with signature `(item, hook)`.
 - `item` The item with all its initial values, plus all of its included items. The function can still reference values which will be later removed by `only` and `exclude`.
 - `hook` The hook passed to serialize.

Serialize examples

A simple serialize

The populate [example above](#) produced the result

```
{ id: 9, title: 'The unbearable ligtness of FeathersJS', author: 5, yearBorn: 1990,
  authorItem: { id: 5, email: 'john.doe@gmail.com', name: 'John Doe' },
  _include: ['authorItem']
}
```

We could tailor the result more to what we need with:

```
const serializeSchema = {
  only: ['title'],
  authorItem: {
    only: ['name']
    computed: {
      isOver18: (authorItem, hook) => new Date().getFullYear() - authorItem.yearBorn >= 18,
    },
  },
};
app.service('posts').before({
  get: [ hooks.populate({ schema }), serialize(serializeSchema) ],
  find: [ hooks.populate({ schema }), serialize(serializeSchema) ]
});
```

The result would now be

```
{ title: 'The unbearable ligtness of FeathersJS',
  authorItem: { name: 'John Doe', isOver18: true, _computed: ['isOver18'] },
  _include: ['authorItem'],
}
```

Using permissions

Consider an Employee item. The Payroll Manager would be permitted to see the salaries of other department heads. No other person would be allowed to see them.

Using a function for `schema` allows you to select an appropriate schema based on the need.

Assume `hook.params.user.roles` contains an array of roles which the user performs. The Employee item can be serialized differently for the Payroll Manager than for anyone else.

```
const payrollSerialize = {
  'payrollMgr': { /* serialization schema for Payroll Manager */},
  'payroll': { /* serialization schema for others */}
};

employees.after({
  all: [
    populate( ... ),
    serialize(hook => payrollSerialize[
      hook.params.user.roles.contains('payrollMgr') ? 'payrollMgr' : 'payroll'
    ])
  ]
});
```

```
]);
```

setCreatedAt

setCreatedAt(fieldName = 'createdAt', ... fieldNames) [source](#)

Add the fields with the current date-time.

- Used as a `before` hook for `create`, `update` or `patch`.
- Used as an `after` hook.
- Field names support dot notation.
- Supports multiple data items, including paginated `find`.

ProTip `setCreatedAt` will be deprecated, so use `setNow` instead.

```
const { setCreatedAt } = require('feathers-hooks-common');

// set the `createdAt` field before a user is created
app.service('users').before({
  create: [ setCreatedAt() ]
});
```

Options:

- `fieldName` (*optional*, default: `createdAt`) - The field that you want to add with the current date-time to the retrieved object(s).
- `fieldNames` (*optional*) - Other fields to add with the current date-time.

See also `setUpdatedAt`.

setNow

setNow(... fieldNames) [source](#)

Add the fields with the current date-time.

- Used as a `before` hook for `create`, `update` or `patch`.
- Used as an `after` hook.
- Field names support dot notation.
- Supports multiple data items, including paginated `find`.

```
const { setNow } = require('feathers-hooks-common');

app.service('users').before({
  create: setNow('createdAt', 'updatedAt')
});
```

Options:

- `fieldNames` (*required*, at least one) - The fields that you want to add with the current date-time to the retrieved object(s).

ProTip Use `setNow` rather than `setCreatedAt` or `setUpdatedAt`.

setSlug

setSlug(slug, fieldName = 'query.' + slug) source

A service may have a slug in its URL, e.g. `storeId` in `app.use('/stores/:storeId/candies', new Service())`. The service gets slightly different values depending on the transport used by the client.

transport	<code>hook.data.storeId</code>	<code>hook.params.query</code>	code run on client
socketio	<code>undefined</code>	<code>{ size: 'large', storeId: '123' }</code>	<code>candies.create({ name: 'Gummi', qty: 100 }, { query: { size: 'large', storeId: '123' } })</code>
rest	<code>:storeId</code>	... same as above	... same as above
raw HTTP	<code>123</code>	<code>{ size: 'large' }</code>	<code>fetch('/stores/123/candies?size=large', ...)</code>

This hook normalizes the difference between the transports. A hook of `all: [hooks.setSlug('storeId')]` provides a normalized `hook.params.query` of `{ size: 'large', storeId: '123' }` for the above cases.

- Used as a `before` hook.
- Field names support dot notation.

```
const { setSlug } = require('feathers-hooks-common');

app.service('stores').before({
  create: [ setSlug('storeId') ]
});
```

Options:

- `slug` (*required*) - The slug as it appears in the route, e.g. `storeId` for `/stores/:storeId/candies`.
- `fieldName` (*optional*, default: `query[slugId]`) - The field to contain the slug value.

setUpdatedAt

setUpdatedAt(fieldName = 'updatedAt', ...fieldNames) source

Add or update the fields with the current date-time.

- Used as a `before` hook for `create`, `update` OR `patch`.
- Used as an `after` hook.
- Field names support dot notation.
- Supports multiple data items, including paginated `find`.

ProTip `setUpdatedAt` will be deprecated, so use `setNow` instead.

```
const { setUpdatedAt } = require('feathers-hooks-common');

// set the `updatedAt` field before a user is created
app.service('users').before({
  create: [ setUpdatedAt() ]
});
```

Options:

- `fieldName` (*optional*, default: `updatedAt`) - The fields that you want to add or update in the retrieved object(s).
- `fieldNames` (*optional*) - Other fields to add or update with the current date-time.

See also `setCreatedAt`.

sifter

sifter(mongoQueryFunc) [source](#)

All official Feathers database adapters support a common way for querying, sorting, limiting and selecting find method calls. These are limited to what is commonly supported by all the databases.

The `sifter` hook provides an extensive MongoDB-like selection capabilities, and it may be used to more extensively select records.

- Used as an `after` hook for `find`.
- Provides extensive MongoDB-like selection capabilities.

ProTip `sifter` filters the result of a `find` call. Therefore more records will be physically read than needed.

You can use the Feathers database adapters `query` to reduce this number.

```
const sift = require('sift');
const { sifter } = require('feathers-hooks-common');

const selectCountry = hook => sift({ 'address.country': hook.params.country });

app.service('stores').after({
  find: sifter(selectCountry),
});

const sift = require('sift');
const { sifter } = require('feathers-hooks-common');

const selectCountry = country => () => sift({ address: { country: country } });

app.service('stores').after({
  find: sifter(selectCountry('Canada')),
});
```

Options:

- `mongoQueryFunc` (*required*) - Function similar to `hook => sift(mongoQueryObj)`. Information about the `mongoQueryObj` syntax is available at [sift](#).

softDelete

softDelete(fieldName = 'deleted') [source](#)

Marks items as `{ deleted: true }` instead of physically removing them. This is useful when you want to discontinue use of, say, a department, but you have historical information which continues to refer to the discontinued department.

- Used as a `before.all` hook to handle all service methods.
- Supports multiple data items, including paginated `find`.

```
const { softDelete } = require('feathers-hooks-common');
const dept = app.service('departments');

dept.before({
  all: softDelete(),
```

```
});

// will throw if item is marked deleted.
dept.get(0).then()

// methods can be run avoiding softDelete handling
dept.get(0, { query: { $disableSoftDelete: true }}).then()
```

Options:

- `fieldName` (*optional*, default: `deleted`) - The name of the field holding the deleted flag.

some

some(. . . hookFuncs) [source](#)

Run hook functions in parallel. Return `true` if any hook function returned a truthy value.

- Used as a predicate function with conditional hooks.
- The current `hook` is passed to all the hook functions, and they are run in parallel.
- Hooks to run may be sync or Promises only.
- `feathers-hooks` catches any errors thrown in the predicate.

```
service.before({
  create: hooks.iff(hooks.some(hook1, hook2, ...), hookA, hookB, ...)
});
```

```
hooks.some(hook1, hook2, ...).call(this, currentHook)
  .then(bool => { ... });
```

Options:

- `hookFuncs` (*required*) Functions which take the current hook as a param and return a boolean result.

See also `every`.

stashBefore

stashBefore(name) [source](#)

Stash current value of record before mutating it.

- Used as a `before` hook for `get` , `update` , `patch` OR `remove` .
- An `id` is required in the method call.

```
service.before({
  patch: stashBefore()
});
```

Options:

- `name` (*optional* defaults to 'before') The name of the params property to contain the current record value.

traverse

traverse(transformer, getObject) [source](#)

Traverse and transform objects in place by visiting every node on a recursive walk.

- Used as a `before` or `after` hook.
- Supports multiple data items, including paginated `find`.
- Any object in the hook may be traversed, **including the query object**.
- `transformer` has access to powerful methods and context.

```
// Trim strings
const trimmer = function (node) {
  if (typeof node === 'string') { this.update(node.trim()); }
};
service.before({ create: traverse(trimmer) });
```

```
// REST HTTP request may use the string 'null' in its query string.
// Replace these strings with the value null.
const nullder = function (node) {
  if (node === 'null') { this.update(null); }
};
service.before({ find: traverse(nullder, hook => hook.params.query) });
```

ProTip: GitHub's [substack/js-traverse](#) documents the extensive methods and context available to the transformer function.

Options:

- `transformer` (*required*) - Called for every node and may change it in place.
- `getObject` (*optional*, defaults to `hook.data` or `hook.result`) - Function with signature (hook) which returns the object to traverse.

unless

unless(predicate, ...hookFuncs) [source](#)

Resolve the predicate to a boolean. Run the hooks sequentially if the result is falsey.

- Used as a `before` or `after` hook.
- Predicate may be a sync or async function.
- Hooks to run may be sync, Promises or callbacks.
- `feathers-hooks` catches any errors thrown in the predicate or hook.

```
service.before({
  create:
    unless(isProvider('server'),
      hookA,
      unless(isProvider('rest'), hook1, hook2, hook3),
      hookB
    )
});
```

Options:

- `predicate` (*required*) - Determines if hookFuncs should be run or not. If a function, `predicate` is called with the hook as its param. It returns either a boolean or a Promise that evaluates to a boolean.
- `hookFuncs` (*optional*) - Zero or more hook functions. They may include other conditional hook functions.

See also `iff`, `iffElse`, `else`, `when`, `isNot`, `isProvider`.

validate

validate(`validator`) [source](#)

Call a validation function from a `before` hook. The function may be sync or return a Promise.

- Used as a `before` hook for `create`, `update` or `patch`.

ProTip: If you have a different signature for the validator then pass a wrapper as the validator e.g. `(values) => myValidator(..., values, ...)`.

ProTip: Wrap your validator in `callbackToPromise` if it uses a callback.

```
const { callbackToPromise, validate } = require('feathers-hooks-common');

// function myCallbackValidator(values, cb) { ... }
const myValidator = callbackToPromise(myCallbackValidator, 1); // function requires 1 param
app.service('users').before({ create: validate(myValidator) });
```

Options:

- `validator (required)` - Validation function with signature `function validator(formValues, hook)`.

Sync functions return either an error object like `{ fieldName1: 'message', ... }` or null. Validate will throw on an error object with `throw new errors.BadRequest({ errors: errorObject })`.

Promise functions should throw on an error or reject with `new errors.BadRequest('Error message', { errors: { fieldName1: 'message', ... } })`. Their `.then` returns either sanitized values to replace `hook.data`, or null.

Example

Comprehensive validation may include the following:

- Object schema validation. Checking the item object contains the expected properties with values in the expected format. The values might get sanitized. Knowing the item is well formed makes further validation simpler.
- Re-running any validation supposedly already done on the front-end. It would be an asset if the server can re-run the same code the front-end used.
- Performing any validation and sanitization unique to the server.

A full featured example of such a process appears below. It validates and sanitizes a new user before adding the user to the database.

- The form expects to be notified of errors in the format `{ email: 'Invalid email.', password: 'Password must be at least 8 characters.' }`.
- The form calls the server for async checking of selected fields when control leaves those fields. This for example could check that an email address is not already used by another user.
- The form does local sync validation when the form is submitted.
- The code performing the validations on the front-end is also used by the server.
- The server performs schema validation using Walmart's [Joi](#).
- The server does further validation and sanitization.

Validation using Validate

```
// file /server/services/users/hooks/index.js
const auth = require('feathers-authentication').hooks;
const { callbackToPromise, remove, validate } = require('feathers-hooks-common');
const validateSchema = require('feathers-hooks-validate-joi');

const clientValidations = require('/common/usersClientValidations');
const serverValidations = require('/server/validations/usersServerValidations');
const schemas = require('/server/validations/schemas');

const serverValidationsSignup = callbackToPromise(serverValidations.signup, 1);

exports.before = {
  create: [
    validateSchema.form(schemas.signup, schemas.options), // schema validation
    validate(clientValidations.signup), // re-run form sync validation
    validate(values => clientValidations.signupAsync(values, 'someMoreParams')), // re-run form async
    validate(serverValidationsSignup), // run server validation
    remove('confirmPassword'),
    auth.hashPassword()
  ]
};


```

Validation routines for front and back-end.

Validations used on front-end. They are re-run by the server.

```
// file /common/usersClientValidations
// Validations for front-end. Also re-run on server.
const clientValidations = {};

// sync validation of signup form on form submit
clientValidations.signup = values => {
  const errors = {};

  checkName(values.name, errors);
  checkUsername(values.username, errors);
  checkEmail(values.email, errors);
  checkPassword(values.password, errors);
  checkConfirmPassword(values.password, values.confirmPassword, errors);

  return errors;
};

// async validation on exit from some fields on form
clientValidations.signupAsync = values =>
  new Promise((resolve, reject) => {
    const errs = {};

    // set a dummy error
    errs.email = 'Already taken.';

    if (!Object.keys(errs).length) {
      resolve(null); // 'null' as we did not sanitize 'values'
    }
    reject(new errors.BadRequest('Values already taken.', { errors: errs }));
  });

module.exports = clientValidations;

function checkName(name, errors, fieldName = 'name') {
  if (!/^[\sa-zA-Z]{8,30}$/.test((name || '').trim())) {
    errors[fieldName] = 'Name must be 8 or more letters or spaces.';
  }
}
```

Schema definitions used by the server.

```
// file /server/validations/schemas
const Joi = require('joi');

const username = Joi.string().trim().alphanum().min(5).max(30).required();
const password = Joi.string().trim().regex(/^[a-zA-Z0-9]+$/, 'letters, numbers, spaces')
  .min(8).max(30).required();
const email = Joi.string().trim().email().required();

module.exports = {
  options: { abortEarly: false, convert: true, allowUnknown: false, stripUnknown: true },
  signup: Joi.object().keys({
    name: Joi.string().trim().min(8).max(30).required(),
    username,
    password,
    confirmPassword: password.label('Confirm password'),
    email
  })
};
```

Validations run by the server.

```
// file /server/validations/usersServerValidations
// Validations on server. A callback function is used to show how the hook handles it.
module.exports = {
  signup: (data, cb) => {
    const formErrors = {};
    const sanitized = {};

    Object.keys(data).forEach(key => {
      sanitized[key] = (data[key] || '').trim();
    });

    cb(Object.keys(formErrors).length > 0 ? formErrors : null, sanitized);
  }
};
```

See also validateSchema.

validateSchema

validateSchema(schema, ajv, options) [source](#)

Validate an object using [JSON-Schema](#) through [AJV](#)

ProTip There are some [good tutorials](#) on using JSON-Schema with [ajv](#).

- Used as a `before` or `after` hook.
- The hook will throw if the data does not match the JSON-Schema. `error.errors` will, by default, contain an array of error messages.

ProTip You may customize the error message format with a custom formatting function. You could, for example, return `{ name1: message, name2: message }` which could be more suitable for a UI.

ProTip If you need to customize `ajv` with new keywords, formats or schemas, then instead of passing the `Ajv` constructor, you may pass in an instance of `Ajv` as the second parameter. In this case you need to pass `ajv` options to the `ajv` instance when `new`ing, rather than passing them in the third parameter of `validateSchema`. See the second example below.

```
const Ajv = require('ajv');
const createSchema = { /* JSON-Schema */ };
module.before({
  create: validateSchema(createSchema, Ajv)
});
```

```
const Ajv = require('ajv');
const ajv = new Ajv({ allErrors: true, $data: true });
ajv.addFormat('allNumbers', '^\\d+$');
const createSchema = { /* JSON-Schema */ };
module.before({
  create: validateSchema(createSchema, ajv)
});
```

Options:

- `schema` (*required*) - The JSON-Schema.
- `ajv` (*required*) - The `ajv` validator. Could be either the `Ajv` constructor or an instance of it.
- `options` (*optional*) - Options.
 - Any `ajv` options. Only effective when the second parameter is the `Ajv` constructor.
 - `addNewError` (*optional*) - Custom message formatter. Its a reducing function which works similarly to `Array.reduce()`. Its signature is `{ currentFormattedMessages: any, ajvError: AjvError, itemsLen: number, index: number } : newFormattedMessages`
 - `currentFormattedMessages` - Formatted messages so far. Initially `null`.
 - `ajvError` - `ajv` error.
 - `itemsLen` - How many data items there are. 1-based.
 - `index` - Which item this is. 0-based.
 - `newFormattedMessages` - The function returns the updated formatted messages.

ProTip: You can consider using `ajv-i18n`, together with the `messages` option, to internationalize your error messages.

when

An alias for `iff source`

Util: callbackToPromise**callbackToPromise(callbackFunc, paramsCount) source**

Wrap a function calling a callback into one that returns a Promise.

- Promise is rejected if the function throws.

```
const { callbackToPromise } = require('feathers-hooks-common');

function tester(data, a, b, cb) {
  if (data === 3) { throw new Error('error thrown'); }
  cb(data === 1 ? null : 'bad', data);
}
const wrappedTester = callbackToPromise(tester, 3); // because func call requires 3 params

wrappedTester(1, 2, 3); // tester(1, 2, 3, wrapperCb)
wrappedTester(1, 2); // tester(1, 2, undefined, wrapperCb)
wrappedTester(); // tester(undefined, undefined undefined, wrapperCb)
wrappedTester(1, 2, 3, 4, 5); // tester(1, 2, 3, 4, 5, wrapperCb)
```

```
wrappedTester(1, 2, 3).then( ... )
  .catch(err => { console.log(err instanceof Error ? err.message : err); });
```

Options:

- `callbackFunc` (*required*) - A function which uses a callback as its last param.
- `paramsCount` (*required*) - The number of parameters `callbackFunc` expects. This count does not include the callback param itself.

The wrapped function will always be called with that many params, preventing potential bugs.

See also `promiseToCallback`.

Util: checkContext

`checkContext(hook, type, methods, label)` [source](#)

Restrict the hook to a hook type (before, after) and a set of hook methods (find, get, create, update, patch, remove).

```
const { checkContext } = require('feathers-hooks-common');

function myHook(hook) {
  checkContext(hook, 'before', ['create', 'remove']);
  ...
}

app.service('users').after({
  create: [ myHook ] // throws
});

// checkContext(hook, 'before', ['update', 'patch'], 'hookName');
// checkContext(hook, null, ['update', 'patch']);
// checkContext(hook, 'before', null, 'hookName');
// checkContext(hook, 'before');
```

Options:

- `hook` (*required*) - The hook provided to the hook function.
- `type` (*optional*) - The hook may be run in `before` or `after`. `null` allows the hook to be run in either.
- `methods` (*optional*) - The hook may be run for these methods.
- `label` (*optional*) - The label to identify the hook in error messages, e.g. its name.

Util: deleteByDot

`deleteByDot(obj, path)` [source](#)

`deleteByDot` deletes a property from an object using dot notation, e.g. `employee.address.city`.

```
import { deleteByDot } from 'feathers-hooks-common';

const discardPasscode = () => (hook) => {
  deleteByDot(hook.data, 'security.passcode');
}

app.service('directories').before = {
  find: discardPasscode()
};
```

Options:

- `obj` (*required*) - The object containing the property we want to delete.
- `path` (*required*) - The path to the data, e.g. `security.passcode`. Array notion is *not* supported, e.g. `order.lineItems[1].quantity`.

See also `existsByDot`, `getByDot`, `setByDot`.

Util: `existsByDot`

`existsByDot(obj, path)` source

`existsByDot` checks if a property exists in an object using dot notation, e.g. `employee.address.city`. Properties with a value of `undefined` are considered to exist.

```
import { discard, existsByDot, iff } from 'feathers-hooks-common';

const discardBadge = () => iff(!existsByDot('security.passcode'), discard('security.badge'));

app.service('directories').before = {
  find: discardBadge()
};
```

Options:

- `obj` (*required*) - The object containing the property.
- `path` (*required*) - The path to the property, e.g. `security.passcode`. Array notion is *not* supported, e.g. `order.lineItems[1].quantity`.

See also `existsByDot`, `getByDot`, `setByDot`.

Util: `getByDot`, `setByDot`

`getByDot(obj, path)` source

`setByDot(obj, path, value, ifDelete)` source

`getByDot` gets a value from an object using dot notation, e.g. `employee.address.city`. It does not differentiate between non-existent paths and a value of `undefined`.

`setByDot` is the companion to `getByDot`. It sets a value in an object using dot notation.

```
import { getByDot, setByDot } from 'feathers-hooks-common';

const setHomeCity = () => (hook) => {
  const city = getByDot(hook.data, 'person.address.city');
  setByDot(hook, 'data.person.home.city', city);
}

app.service('directories').before = {
  create: setHomeCity()
};
```

Options:

- `obj` (*required*) - The object we get data from or set data in.
- `path` (*required*) - The path to the data, e.g. `person.address.city`. Array notion is *not* supported, e.g. `order.lineItems[1].quantity`.
- `value` (*required*) - The value to set the data to.

See also `existsByDot`, `deleteByDot`.

Util: `getItems`, `replaceItems`

getItems(hook) [source](#)

replaceItems(hook, items) [source](#)

`getItems` gets the data items in a hook. The items may be `hook.data`, `hook.result` OR `hook.result.data` depending on where the hook is used, the method its used with and if pagination is used. `undefined`, an object or an array of objects may be returned.

`replaceItems` is the companion to `getItems`. It updates the data items in the hook.

- Handles before and after hooks.
- Handles paginated and non-paginated results from `find`.

```
import { getItems, replaceItems } from 'feathers-hooks-common';

const insertCode = (code) => (hook) {
  const items = getItems(hook);
  !Array.isArray(items) ? items.code = code : (items.forEach(item => { item.code = code; }));
  replaceItems(hook, items);
}

app.service('messages').before = {
  create: insertCode('a')
};
```

The common hooks usually mutate the items in place, so a `replaceItems` is not required.

```
const items = getItems(hook);
(Array.isArray(items) ? items : [items]).forEach(item => { item.setCreatedAt = new Date(); });
```

Options:

- `hook` (*required*) - The hook provided to the hook function.
- `items` (*required*) - The updated item or array of items.

Util: `paramsForServer`

paramsForServer(params, ... whitelist) [source](#)

A client utility to pass selected `params` properties to the server.

- Companion to the server-side hook `paramsFromClient`.

By default, only the `hook.params.query` object is transferred to the server from a Feathers client, for security among other reasons. However you can explicitly transfer other `params` props with the client utility function `paramsForServer` in conjunction with the hook function `paramsFromClient` on the server.

```
// client
import { paramsForServer } from 'feathers-hooks-common';
service.patch(null, data, paramsForServer({
  query: { dept: 'a' }, populate: 'po-1', serialize: 'po-mgr'
}));

// server
const { paramsFromClient } = require('feathers-hooks-common');
service.before({ all: [
  paramsFromClient('populate', 'serialize', 'otherProp'),
  myHook
]});

// myHook's `hook.params` will now be
// { query: { dept: 'a' }, populate: 'po-1', serialize: 'po-mgr' } }
```

Options:

- `params` (*optional*) The `params` object to pass to the server, including any `query` prop.
- `whitelist` (*optional*) Names of the props in `params` to transfer to the server. This is a security feature. All props are transferred if no whitelist is specified.

See `paramsFromClient`.

Util: promiseToCallback

promiseToCallback(promise)(callbackFunc) [source](#)

Wrap a Promise into a function that calls a callback.

- The callback does not run in the Promise's scope. The Promise's `catch` chain is not invoked if the callback throws.

```
import { promiseToCallback } from 'feathers-hooks-common'

function (cb) {
  const promise = new Promise( ... ).then( ... ).catch( ... );
  ...
  promiseToCallback(promise)(cb);
  promise.then( ... ); // this is still possible
}
```

Options:

- `promise` (*required*) - A function returning a promise.

See also `callbackToPromise`.

FAQ: Coerce data types

A common need is converting fields coming in from query params. These fields are provided as string values by default and you may need them as numbers, booleans, etc.

The `validateSchema` does a wide selection of [type coercions](#), as well as checking for missing and unexpected fields.

Client

One of the most notable features of Feathers is that it can also be used as the client. The difference to many other frameworks and services is that it isn't a separate library, you instead get the exact same functionality as on the server. This means you can use [services](#) and [hooks](#) and configure plugins. By default a Feathers client automatically creates services that talk to a Feathers server. How to initialize a connection can be found in

- The [REST transport client chapter](#)
- The [Socket.io transport client chapter](#) (real-time)
- The [Primus transport client chapter](#) (real-time)

This chapter describes how to use Feathers as the client in Node, React Native and in the browser with a module loader like Webpack, Browserify, StealJS or through a `<script>` tag.

Important: The Feathers client libraries come transpiled to ES5 but require ES6 shims either through the `babel-polyfill` module or by including `core.js` in older browsers e.g. via `<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>`

Important: If you are not using a module loader you can load the `feathers-client` directly through a script tag in the browser.

Server package	Client package	API page
<code>feathers</code>	<code>feathers/client</code>	Application API
<code>feathers-hooks</code>	<code>feathers-hooks</code>	Hooks API
<code>feathers-errors</code>	<code>feathers-errors</code>	Errors API
<code>feathers-rest</code>	<code>feathers-rest/client</code>	REST Transport API
<code>feathers-socketio</code>	<code>feathers-socketio/client</code>	Socket.io Transport API
<code>feathers-primus</code>	<code>feathers-primus/client</code>	Primus Transport API
<code>feathers-authentication</code>	<code>feathers-authentication-client</code>	Feathers Authentication Client API

Universal (Isomorphic) API

The Feathers Client uses the same [Application API](#) as is available on the server. It is extremely lightweight, however, with Express having been replaced by a [thin wrapper](#). There are differences between the client and server APIs. Learn more under each method on the [Application API page](#).

Node and npm loaders

The client utilities can be used directly on the server. Just `require` each individual package or the `feathers-client` and use it the same way as shown for in the browser examples, below. Node.js natively supports the CommonJS module syntax. Here's an example of setting up the client in Node:

```
const feathers = require('feathers/client');
const socketio = require('feathers-socketio/client');
const hooks = require('feathers-hooks');
const errors = require('feathers-errors'); // An object with all of the custom error types.
const auth = require('feathers-authentication-client');
const io = require('socket.io-client/dist/socket.io');
```

```

const socket = io('http://localhost:3030', {
  transports: ['websocket']
});

const feathersClient = feathers()
  .configure(socketio(socket))
  .configure(hooks())
  .configure(auth())

module.exports = feathersClient;

```

Browserify, StealJS, and Webpack

Both, Browserify and StealJS support npm modules and require no additional configuration. The client modules are all JavaScript, and should also work with any Webpack configuration. Here's the same example from above, rewritten in ES Module syntax:

```

import feathers from 'feathers/client';
import socketio from 'feathers-socketio/client';
import hooks from 'feathers-hooks';
import errors from 'feathers-errors'; // An object with all of the custom error types.
import auth from 'feathers-authentication-client';
import io from 'socket.io-client/dist/socket.io';

const socket = io('http://localhost:3030', {
  transports: ['websocket']
});

const feathersClient = feathers()
  .configure(socketio(socket))
  .configure(hooks())
  .configure(auth())

export default feathersClient;

```

React Native

Install the required packages into your [React Native](#) project.

```
$ npm install feathers feathers-socketio feathers-hooks socket.io-client babel-polyfill
```

Then in the main application file:

```

import 'babel-polyfill';
import io from 'socket.io-client';
import feathers from 'feathers/client';
import socketio from 'feathers-socketio/client';
import hooks from 'feathers-hooks';

const socket = io('http://api.my-feathers-server.com', {
  transports: ['websocket'],
  forceNew: true
});
const app = feathers()
  .configure(hooks())
  .configure(socketio(socket));

const messageService = app.service('messages');

```

```
messageService.on('created', message => console.log('Created a message', message));

// Use the messages service from the server
messageService.create({
  text: 'Message from client'
});
```

feathers-client



```
$ npm install feathers-client --save
```

`feathers-client` is a module that bundles the separate Feathers client side modules into one. It also provides a distributable file with everything you need to use Feathers in the browser through a `<script>` tag. Here is a table of which Feathers client module is included in `feathers-client`:

Feathers module	feathers-client
<code>feathers/client</code>	<code>feathers</code> (default)
<code>feathers-hooks</code>	<code>feathers.hooks</code>
<code>feathers-errors</code>	<code>feathers.errors</code>
<code>feathers-rest/client</code>	<code>feathers.rest</code>
<code>feathers-socketio/client</code>	<code>feathers.socketio</code>
<code>feathers-primus/client</code>	<code>feathers.primus</code>
<code>feathers-authentication/client</code>	<code>feathers.authentication</code>

Load from CDN with `<script>`

Below is an example of the scripts you would use to load `feathers-client` from `unpkg.com`. It's possible to use it with a module loader, but using individual client packages will allow you to take advantage of Feathers' modularity.

```
<script src="//unpkg.com/feathers-client@^2.0.0/dist/feathers.js"></script>
<script src="//unpkg.com/socket.io-client@1.7.3/dist/socket.io.js"></script>
<script>
  // Socket.io is exposed as the `io` global.
  var socket = io('http://localhost:3030', {transports: ['websocket']});
  // feathers-client is exposed as the `feathers` global.
  var feathersClient = feathers()
    .configure(feathers.hooks())
    .configure(feathers.socketio(socket))
    .configure(feathers.authentication())

  // feathers.errors is an object with all of the custom error types.
</script>
```

When to use `feathers-client`

- If you want to use Feathers in the browser with a `<script>` tag
- With a module loader that does not support npm packages (like RequireJS)

You can use `feathers-client` in NodeJS or with a browser module loader/bundler but it will include packages you may not use. It is also important to note that - except for this section - all other Feathers client examples assume you are using Node or a module loader.

RequireJS

Here's an example of loading `feathers-client` using RequireJS Syntax:

```
define(function (require) {
  const feathers = require('feathers-client');
  const socketio = feathers.socketio;
  const hooks = feathers.hooks;
  const errors = feathers.errors; // An object with all of the custom error types.
  const auth = feathers.auth;
  const io = require('socket.io-client');

  const socket = io('http://localhost:3030', {
    transports: ['websocket']
  });

  const feathersClient = feathers()
    .configure(socketio(socket))
    .configure(hooks())
    .configure(auth())

  return feathersClient;
});
```

Events

Events are the key part of Feathers real-time functionality. All events in Feathers are provided through the [NodeJS EventEmitter](#) interface. This section describes

- A quick overview of the [NodeJS EventEmitter interface](#)
- The standard [service events](#)
- How to [filter events](#) so that only allowed clients receive them
- How to allow sending [custom events](#) from the server to the client

Very important: [Event filters](#) are critical for properly securing a Feathers real-time application.

EventEmitters

Once registered, any [service](#) gets turned into a standard [NodeJS EventEmitter](#) and can be used accordingly.

```
const messages = app.service('messages');

// Listen to a normal service event
messages.on('patched', message => console.log('message patched', message));

// Only listen to an event once
messages.once('removed', message =>
  console.log('First time a message has been removed', message)
);

// A reference to a handler
const onCreatedListener = message => console.log('New message created', message);

// Listen `created` with a handler reference
messages.on('created', onCreatedListener);

// Unbind the `created` event listener
messages.removeListener('created', onCreatedListener);

// Send a custom event
messages.emit('customEvent', {
  type: 'customEvent',
  data: 'can be anything'
});
```

Service Events

Any service automatically emits `created`, `updated`, `patched` and `removed` events when the respective service method returns successfully. This works on the client as well as on the server. When the client is using [Socket.io](#) or [Primus](#), events will be pushed automatically from the server to all connected client. This is essentially how Feathers does real-time.

ProTip: Events are not fired until all of your [hooks](#) have executed.

created

The `created` event will fire with the result data when a service `create` returns successfully.

```
const feathers = require('feathers');
```

```

const app = feathers();

app.use('/messages', {
  create(data, params) {
    return Promise.resolve(data);
  }
});

// Retrieve the wrapped service object which will be an event emitter
const messages = app.service('messages');

messages.on('created', message => console.log('created', message));

messages.create({
  text: 'We have to do something!'
});

```

updated, patched

The `updated` and `patched` events will fire with the callback data when a service `update` or `patch` method calls back successfully.

```

const feathers = require('feathers');
const app = feathers();

app.use('/my/messages/', {
  update(id, data) {
    return Promise.resolve(data);
  },
  patch(id, data) {
    return Promise.resolve(data);
  }
});

const messages = app.service('my/messages');

messages.on('updated', message => console.log('updated', message));
messages.on('patched', message => console.log('patched', message));

messages.update(0, {
  text: 'updated message'
});

messages.patch(0, {
  text: 'patched message'
});

```

removed

The `removed` event will fire with the callback data when a service `remove` calls back successfully.

```

const feathers = require('feathers');
const app = feathers();

app.use('/messages', {
  remove(id, params) {
    return Promise.resolve({ id });
  }
});

const messages = app.service('messages');

messages.on('removed', message => console.log('removed', message));

```

```
messages.remove(1);
```

Event Filtering

By default all service events will be sent to **all** connected clients. In many cases you probably want to be able to only send events to certain clients, say maybe only ones that are authenticated or only users that belong to the same company. The [Socket.io](#) and [Primus](#) provider add a `.filter()` service method which can be used to filter events. A filter is a `function(data, connection, hook)` that runs for every connected client and gets passed

- `data` - the data to dispatch.
- `connection` - the connected socket for which the data is being filtered. This is the `feathers` property from the Socket.io and Primus middleware and usually contains information like the connected user.
- `hook` - the hook object from the original method call.

It either returns the data to dispatch or `false` if the event should not be dispatched to this client. Returning a Promise that resolves accordingly is also supported.

ProTip: Filter functions run for every connected client on every event and should be optimized for speed and chained by granularity. That means that general and quick filters should run first to narrow down the connected clients to then run more involved checks if necessary.

Registering filters

There are several ways filter functions can be registered, very similar to how [hooks](#) can be registered.

```
const todos = app.service('todos');

// Register a filter for all events
todos.filter(function(data, connection, hook) {});

// Register a filter for the `created` event
todos.filter('created', function(data, connection, hook) {});

// Register a filter for the `created` and `updated` event
todos.filter({
  created(data, connection, hook) {},
  updated(data, connection, hook) {}
});

// Register a filter chain the `created` and `removed` event
todos.filter({
  created: [ filterA, filterB ],
  removed: [ filterA, filterB ]
});
```

Filter examples

The following example filters all events on the `messages` service if the connection does not have an [authenticated user](#):

```
const messages = app.service('messages');

messages.filter(function(data, connection) {
  if(!connection.user) {
    return false;
  }

  return data;
```

```
});
```

As mentioned, filters can be chained. So once the previous filter passes (the connection has an authenticated user) we can now filter all connections where the data and the user do not belong to the same company:

```
// Blanket filter out all connections that don't belong to the same company
messages.filter(function(data, connection) {
  if(data.company_id !== connection.user.company_id) {
    return false;
  }

  return data;
});
```

Now that we know the connection has an authenticated user and the data and the user belong to the same company, we can filter the `created` event to only be sent if the connections user and the user that created the Message are friends with each other:

```
// After that, filter messages, if the user that created it
// and the connected user aren't friends
messages.filter('created', function(data, connection, hook) {
  // The id of the user that created the todo
  const messageUserId = hook.params.user._id;
  // The a list of ids of the connection's user friends
  const currentUserFriends = connection.user.friends;

  if(currentUserFriends.indexOf(messageUserId) === -1) {
    return false;
  }

  return data;
});
```

Filtering Custom Events

[Custom events](#) can be filtered the same way:

```
app.service('payments').filter('status', function(data, connection, hook) {

});
```

Custom events

By default, real-time clients will only receive the [standard events](#). However, it is possible to define a list of custom events on a service as `service.events` that should also be passed.

Important: The [database adapters](#) also take a list of custom events as an initialization option.

Important: Custom events can only be sent from the server to the client, not the other way (client to server).

[Learn more](#)

For example, a payment service that sends status events to the client while processing a payment could look like this:

```
class PaymentService {
  constructor() {
    this.events = ['status'];
  },
}
```

```
create(data, params) {
  createStripeCustomer(params.user).then(customer => {
    this.emit('status', { status: 'created' });
    return createPayment(data).then(result => {
      this.emit('status', { status: 'completed' });
    });
  });
}
```

Now clients can listen to the `<servicepath> status` event. Custom events can be [filtered](#) just like standard events.

Errors



```
$ npm install feathers-errors --save
```

The `feathers-errors` module contains a set of standard error classes used by all other Feathers modules as well as an [Express error handler](#) to format those - and other - errors and setting the correct HTTP status codes for REST calls.

Feathers errors

The following error types, all of which are instances of `FeathersError` are available:

ProTip: All of the Feathers plugins will automatically emit the appropriate Feathers errors for you. For example, most of the database adapters will already send `Conflict` or `Unprocessable` errors with the validation errors from the ORM.

- `BadRequest` : 400
- `NotAuthenticated` : 401
- `PaymentError` : 402
- `Forbidden` : 403
- `NotFound` : 404
- `MethodNotAllowed` : 405
- `NotAcceptable` : 406
- `Timeout` : 408
- `Conflict` : 409
- `Unprocessable` : 422
- `GeneralError` : 500
- `NotImplemented` : 501
- `Unavailable` : 503

Feathers errors are pretty flexible. They contain the following fields:

- `type` - `FeathersError`
- `name` - The error name (ie. "BadRequest", "ValidationError", etc.)
- `message` - The error message string
- `code` - The HTTP status code
- `className` - A CSS class name that can be handy for styling errors based on the error type. (ie. "bad-request", etc.)
- `data` - An object containing anything you passed to a Feathers error except for the `errors` object.
- `errors` - An object containing whatever was passed to a Feathers error inside `errors`. This is typically validation errors or if you want to group multiple errors together.

ProTip: To convert a Feathers error back to an object call `error.toJSON()`. A normal `console.log` of a JavaScript Error object will not automatically show those additional properties described above (even though they can be accessed directly).

Here are a few ways that you can use them:

```
const errors = require('feathers-errors');
```

```
// If you were to create an error yourself.
const notFound = new errors.NotFound('User does not exist');

// You can wrap existing errors
const existing = new errors.GeneralError(new Error('I exist'));

// You can also pass additional data
const data = new errors.BadRequest('Invalid email', {
  email: 'sergey@google.com'
});

// You can also pass additional data without a message
const dataWithoutMessage = new errors.BadRequest({
  email: 'sergey@google.com'
});

// If you need to pass multiple errors
const validationErrors = new errors.BadRequest('Invalid Parameters', {
  errors: { email: 'Email already taken' }
});

// You can also omit the error message and we'll put in a default one for you
const validationErrors = new errors.BadRequest({
  errors: {
    email: 'Invalid Email'
  }
});
```

Server Side Errors

Promises swallow errors if you forget to add a `catch()` statement. Therefore, you should make sure that you **always** call `.catch()` on your promises. To catch uncaught errors at a global level you can add the code below to your top-most file.

```
process.on('unhandledRejection', (reason, p) => {
  console.log('Unhandled Rejection at: Promise ', p, ' reason: ', reason);
});
```

REST (Express) errors

The separate `feathers-errors/handler` module is an [Express error handler](#) middleware that formats any error response to a REST call as JSON (or HTML if e.g. someone hits our API directly in the browser) and sets the appropriate error code.

ProTip: Because Feathers extends Express you can use any Express compatible [error middleware](#) with Feathers. In fact, the error handler bundled with `feathers-errors` is just a slightly customized one.

Very Important: Just as in Express, the error handler has to be registered *after* all middleware and services.

app.use(handler())

Set up the error handler with the default configuration.

```
const errorHandler = require('feathers-errors/handler');
const app = feathers();

// before starting the app
app.use(errorHandler())
```

app.use(handler(options))

```
const error = require('feathers-errors/handler');
const app = feathers();

// Just like Express your error middleware needs to be
// set up last in your middleware chain.
app.use(error({
  html: function(error, req, res, next) {
    // render your error view with the error object
    res.render('error', error);
  }
}))
```

ProTip: If you want to have the response in json format be sure to set the `Accept` header in your request to `application/json` otherwise the default error handler will return HTML.

The following options can be passed when creating a new localstorage service:

- `html` (Function|Object) [optional] - A custom formatter function or an object that contains the path to your custom html error pages.

ProTip: `html` can also be set to `false` to disable html error pages altogether so that only JSON is returned.

REST

[Star](#) 51 npm v1.8.0 [changelog .md](#)

```
$ npm install feathers-rest --save
```

The `feathers-rest` module allows you to expose and consume `services` through a RESTful API. This means that you can call a service method through the `GET`, `POST`, `PUT`, `PATCH` and `DELETE` HTTP methods:

Service method	HTTP method	Path
<code>.find()</code>	GET	<code>/messages</code>
<code>.get()</code>	GET	<code>/messages/1</code>
<code>.create()</code>	POST	<code>/messages</code>
<code>.update()</code>	PUT	<code>/messages/1</code>
<code>.patch()</code>	PATCH	<code>/messages/1</code>
<code>.remove()</code>	DELETE	<code>/messages/1</code>

Server

To expose services through a RESTful API we will have to configure the `feathers-rest` plugin and provide our own body parser middleware (usually the standard `Express 4 body-parser`) to make REST `.create`, `.update` and `.patch` calls parse the data in the HTTP body. If you would like to add other middleware *before* the REST handler, call `app.use(middleware)` before registering any services.

```
$ npm install body-parser --save
```

Important: For additional information about middleware, routing and how the REST module works with Express middleware see the [Express chapter](#).

ProTip: The body-parser middleware has to be registered *before* any service. Otherwise the service method will throw a `No data provided` OR `First parameter for 'create' must be an object` error.

app.configure(rest())

Configures the transport provider with a standard formatter sending JSON response via `res.json`.

```
const feathers = require('feathers');
const bodyParser = require('body-parser');
const rest = require('feathers-rest');
const app = feathers();

// Turn on JSON parser for REST services
app.use(bodyParser.json())
// Turn on URL-encoded parser for REST services
app.use(bodyParser.urlencoded({ extended: true }));
// Set up REST transport
app.configure(rest())
```

app.configure(rest(formatter))

The default REST response formatter is a middleware that formats the data retrieved by the service as JSON. If you would like to configure your own `formatter` middleware pass a `formatter(req, res)` function. This middleware will have access to `res.data` which is the data returned by the service. `res.format` can be used for content negotiation.

```
const app = feathers();
const bodyParser = require('body-parser');
const rest = require('feathers-rest');

// Turn on JSON parser for REST services
app.use(bodyParser.json())
// Turn on URL-encoded parser for REST services
app.use(bodyParser.urlencoded({ extended: true }));
// Set up REST transport
app.configure(rest(function(req, res) {
  // Format the message as text/plain
  res.format({
    'text/plain': function() {
      res.end(`The Message is: "${res.data.text}"`);
    }
  });
}));
```

params.query

`params.query` will contain the URL query parameters sent from the client. For the REST transport the query string is parsed using the `qs` module. For some query string examples see the [database querying](#) chapter.

Important: Only `params.query` is passed between the server and the client, other parts of `params` are not. This is for security reasons so that a client can't set things like `params.user` or the database options. You can always map from `params.query` to other `params` properties in a before [hook](#).

params.provider

For any [service method call](#) made through REST `params.provider` will be set to `rest`. In a [hook](#) this can for example be used to prevent external users from making a service method call:

```
app.service('users').hooks({
  before: [
    remove(hook) {
      // check for if(hook.params.provider) to prevent any external call
      if(hook.params.provider === 'rest') {
        throw new Error('You can not delete a user via REST');
      }
    }
  ]
});
```

Client

The `client` module in `feathers-rest` (`require('feathers-rest/client')`) allows to connect to a service exposed through the REST server using [jQuery](#), [request](#), [Superagent](#), [Axios](#) or [Fetch](#) as the AJAX library.

Very important: The examples below assume you are using Feathers either in Node or in the browser with a module loader like Webpack or Browserify. For using Feathers with a `<script>` tag, AMD modules or with React Native see the [client chapter](#).

ProTip: REST client services do emit `created`, `updated`, `patched` and `removed` events but only *locally for their own instance*. Real-time events from other clients can only be received by using a websocket connection.

Note: A client application can only use a single transport (either REST, Socket.io or Primus). Using two transports in the same client application is normally not necessary.

rest([baseUrl])

REST client services can be initialized by loading `feathers-rest/client` and initializing a client object with a base URL:

```
const feathers = require('feathers/client');
const rest = require('feathers-rest/client');

// Connect to REST endpoints
const restClient = rest();
// Connect to a different URL
const restClient = rest('http://feathers-api.com');
```

ProTip: The base URL is relative from where services are registered. That means that a service at `http://api.feathersjs.com/api/v1/messages` with a base URL of `http://api.feathersjs.com` would be available as `app.service('api/v1/messages')`. With a base URL of `http://api.feathersjs.com/api/v1` it would be `app.service('messages')`.

REST client wrappers are always initialized using a base URL:

```
app.configure(restClient.superagent(superagent [, options]));
```

Default headers can be set for all libraries (except `request` which has its own defaults mechanism) in the options like this:

```
app.configure(restClient.superagent(superagent, {
  headers: { 'X-Requested-With': 'FeathersJS' }
}));
```

Then services that automatically connect to that remote URL can be retrieved as usual via `app.service`:

```
app.service('messages').create({
  text: 'A message from a REST client'
});
```

Request specific headers can be through `params.headers` in a service call:

```
app.service('messages').create({
  text: 'A message from a REST client'
}, {
  headers: { 'X-Requested-With': 'FeathersJS' }
});
```

The supported AJAX libraries can be initialized as follows.

jQuery

Pass the instance of jQuery (`$`) to `restClient.jquery`:

```
app.configure(restClient.jquery(window.jQuery));
```

Or with a module loader:

```
import $ from 'jquery';

app.configure(restClient.jquery($));
```

Request

The `request` object needs to be passed explicitly to `feathers.request`. Using `request.defaults` - which creates a new request object - is a great way to set things like default headers or authentication information:

```
const request = require('request');
const client = request.defaults({
  'auth': {
    'user': 'username',
    'pass': 'password',
    'sendImmediately': false
  }
});

app.configure(restClient.request(client));
```

Superagent

[Superagent](#) currently works with a default configuration:

```
const superagent = require('superagent');

app.configure(restClient.superagent(superagent));
```

Axios

[Axios](#) currently works with a default configuration:

```
const axios = require('axios');

app.configure(restClient.axios(axios));
```

Fetch

Fetch also uses a default configuration:

```
const fetch = require('node-fetch');

app.configure(restClient.fetch(fetch));
```

Direct connection

You can communicate with a Feathers server using any HTTP REST client. The following section describes what HTTP method, body and query parameters belong to which service method call.

All query parameters in a URL will be set as `params.query` on the server. Other service parameters can be set through [hooks](#) and [Express middleware](#). URL query parameter values will always be strings. Conversion (e.g. the string `'true'` to boolean `true`) can be done in a hook as well.

The body type for `POST`, `PUT` and `PATCH` requests is determined by the Express [body-parser](#) middleware which has to be registered *before* any service. You should also make sure you are setting your `Accept` header to `application/json`.

find

Retrieves a list of all matching resources from the service

```
GET /messages?status=read&user=10
```

Will call `messages.find({ query: { status: 'read', user: '10' } })` on the server.

If you want to use any of the built-in find operands (`$le`, `$lt`, `$ne`, `$eq`, `$in`, etc.) the general format is as follows:

```
GET /messages?field[$operand]=value&field[$operand]=value2
```

For example, to find the records where field `status` is not equal to `active` you could do

```
GET /messages?status[$ne]=active
```

More information about the possible parameters for official database adapters can be found [in the database querying section](#).

get

Retrieve a single resource from the service.

```
GET /messages/1
```

Will call `messages.get(1, {})` on the server.

```
GET /messages/1?fetch=all
```

Will call `messages.get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
POST /messages
{ "text": "I really have to iron" }
```

Will call `messages.create({ "text": "I really have to iron" }, {})` on the server.

```
POST /messages
[
  { "text": "I really have to iron" },
  { "text": "Do laundry" }
]
```

update

Completely replace a single or multiple resources.

```
PUT /messages/2
{ "text": "I really have to do laundry" }
```

Will call `messages.update(2, { "text": "I really have to do laundry" }, {})` on the server. When no `id` is given by sending the request directly to the endpoint something like:

```
PUT /messages?complete=false
{ "complete": true }
```

Will call `messages.update(null, { "complete": true }, { query: { complete: 'false' } })` on the server.

ProTip: `update` is normally expected to replace an entire resource which is why the database adapters only support `patch` for multiple records.

patch

Merge the existing data of a single or multiple resources with the new `data`.

```
PATCH /messages/2
{ "read": true }
```

Will call `messages.patch(2, { "read": true }, {})` on the server. When no `id` is given by sending the request directly to the endpoint something like:

```
PATCH /messages?complete=false
{ "complete": true }
```

Will call `messages.patch(null, { complete: true }, { query: { complete: 'false' } })` on the server to change the status for all read messages.

This is supported out of the box by the Feathers [database adapters](#)

remove

Remove a single or multiple resources:

```
DELETE /messages/2?cascade=true
```

Will call `messages.remove(2, { query: { cascade: 'true' } })`.

When no `id` is given by sending the request directly to the endpoint something like:

```
DELETE /messages?read=true
```

Will call `messages.remove(null, { query: { read: 'true' } })` to delete all read messages.

Express

On the server, a Feathers application acts as a drop-in replacement for any [Express](#) application. This chapter describes how [services](#) and the [REST transport](#) interact with Express middleware.

Important: This chapter assumes that you are familiar with [Express](#).

Setting service params

All middleware registered after the [REST transport](#) will have access to the `req.feathers` object to set properties on the service method `params`:

```
const app = require('feathers')();
const rest = require('feathers-rest');
const bodyParser = require('body-parser');

app.configure(rest())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({extended: true}))
  .use(function(req, res, next) {
    req.feathers.fromMiddleware = 'Hello world';
    next();
});

app.use('/todos', {
  get(id, params) {
    console.log(params.provider); // -> 'rest'
    console.log(params.fromMiddleware); // -> 'Hello world'

    return Promise.resolve({
      id, params,
      description: `You have to do ${id}!`
    });
  }
});

app.listen(3030);
```

You can see the parameters set by running the example and visiting <http://localhost:3030/todos/test>.

Avoid setting `req.feathers = something` directly since it may already contain information that other Feathers plugins rely on. Adding individual properties or using `Object.assign(req.feathers, something)` is the more reliable option.

Very important: Since the order of Express middleware matters, any middleware that sets service parameters has to be registered *before* your services (in a generated application before `app.configure(services)` or in `middleware/index.js`).

ProTip: Although it may be convenient to set `req.feathers.req = req;` to have access to the request object in the service, we recommend keeping your services as provider independent as possible. There usually is a way to pre-process your data in a middleware so that the service does not need to know about the HTTP request or response.

Query parameters

The query string is parsed using the `qs` module. URL query parameters will be parsed and passed to the service as `params.query`. For example:

```
GET /messages?read=true&$sort[createdAt]=-1
```

Will set `params.query` to

```
{
  "read": "true",
  "$sort": { "createdAt": "-1" }
}
```

For additional query string examples see the [database querying](#) chapter.

ProTip: Since the URL is just a string, there will be **no type conversion**. This can be done manually in a [hook](#).

ProTip: If an array in your request consists of more than 20 items, the `qs` parser implicitly **converts** it to an object with indices as keys. To extend this limit, you can set a custom query parser: `app.set('query parser', str => qs.parse(str, {arrayLimit: 1000}))`

Route parameters

Express route placeholder parameters in a service URL will be added to the service `params`:

```
const feathers = require('feathers');
const rest = require('feathers-rest');

const app = feathers();

app.configure(rest())
  .use(function(req, res, next) {
    req.feathers.fromMiddleware = 'Hello world';
    next();
  });

app.use('/users/:userId/messages', {
  get(id, params) {
    console.log(params.query); // -> ?query
    console.log(params.provider); // -> 'rest'
    console.log(params.fromMiddleware); // -> 'Hello world'
    console.log(params.userId); // will be `1` for GET /users/1/messages

    return Promise.resolve({
      id,
      params,
      read: false,
      text: `Feathers is great!`,
      createdAt: new Date().getTime()
    });
  }
});

app.listen(3030);
```

You can see all the passed parameters by going to something like `localhost:3030/users/213/messages/23?read=false&$sort[createdAt]=-1`.

Custom service middleware

Custom Express middleware that only should run before or after a specific service can be passed to `app.use` in the order it should run:

```
const todoService = {
  get(id) {
    return Promise.resolve({
      id,
      description: `You have to do ${id}!`
    });
  }
};

app.use('/todos', ensureAuthenticated, logRequest, todoService, updateData);
```

Middleware that runs after the service will have `res.data` available which is the data returned by the service. For example `updateData` could look like this:

```
function updateData(req, res, next) {
  res.data.updateData = true;
  next();
}
```

Information about how to use a custom formatter (e.g. to send something other than JSON) can be found in the [REST transport](#) chapter.

Sub-Apps

Sub-apps allow to provide different versions for an API. Currently, when using the Socket.io and Primus [real-time providers](#) providers, `app.setup` will be called automatically, however, with only the REST provider or when using plain Express in the parent application you will have to call the sub-apps `setup` yourself:

```
const express = require('express');
const feathers = require('feathers');
const api = feathers().use('/service', myService);

const mainApp = express().use('/api/v1', api);

const server = mainApp.listen(3030);

// Now call setup on the Feathers app with the server
api.setup(server);
```

ProTip: We recommend avoiding complex sub-app setups because websockets and Feathers built in authentication are not fully sub-app aware.

HTTPS

With your Feathers application initialized it is easy to set up an HTTPS REST and SocketIO server:

```
const fs = require('fs');
const https = require('https');

app.configure(socketio()).use('/todos', todoService);

const server = https.createServer({
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem')
```

```
}, app).listen(443);

// Call app.setup to initialize all services and SocketIO
app.setup(server);
```

Virtual Hosts

You can use the `vhost` middleware to run your Feathers app on a virtual host:

```
const vhost = require('vhost');

app.use('/todos', todoService);

const host = feathers().use(vhost('foo.com', app));
const server = host.listen(8080);

// Here we need to call app.setup because .listen on our virtual hosted
// app is never called
app.setup(server);
```

Socket.io

[Star 27](#) [npm v2.0.0](#) [changelog .md](#)

```
$ npm install feathers-socketio --save
```

The [feathers-socketio](#) module allows to call [service methods](#) and receive [real-time events](#) via [Socket.io](#), a NodeJS library which enables real-time bi-directional, event-based communication.

Service method	Method event name	Real-time event
.find()	messages::find	-
.get()	messages::get	-
.create()	messages::create	messages created
.update()	messages::update	messages updated
.patch()	messages::patch	messages patched
.remove()	messages::removed	messages removed

Important: Socket.io is also used to *call* service methods. Using sockets for both, calling methods and receiving real-time events is generally faster than using [REST](#) and there is usually no need to use both, REST and Socket.io in the same client application at the same time.

Server

app.configure(socketio())

Sets up the Socket.io transport with the default configuration using either the server provided by [app.listen](#) or passed in [app.setup\(server\)](#).

```
const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers();

app.configure(socketio());

app.listen(3030);
```

Pro tip: Once the server has been started with `app.listen()` or `app.setup(server)` the Socket.io object is available as `app.io`.

app.configure(socketio(callback))

Sets up the Socket.io transport with the default configuration and call `callback` with the [Socket.io server object](#). This is a good place to listen to custom events or add [authorization](#):

```
const feathers = require('feathers');
const socketio = require('feathers-socketio');
```

```

const app = feathers();

app.configure(socketio(function(io) {
  io.on('connection', function(socket) {
    socket.emit('news', { text: 'A client connected!' });
    socket.on('my other event', function (data) {
      console.log(data);
    });
  });
});

// Registering Socket.io middleware
io.use(function (socket, next) {
  // Exposing a request property to services and hooks
  socket.feathers.referrer = socket.request.referrer;
  next();
});
)));

app.listen(3030);

```

app.configure(socketio(options [, callback]))

Sets up the Socket.io transport with the given [Socket.io options object](#) and optionally calls the callback described above.

This can be used to e.g. configure the path where Socket.io is initialize (`socket.io/` by default). The following changes the path to `/ws/`:

```

const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers()
  .configure(socketio({
    path: '/ws/'
  }, function(io) {
    // Do something here
    // This function is optional
  }));
);

app.listen(3030);

```

app.configure(socketio(port, [options], [callback]))

Creates a new Socket.io server on a separate port. Options and a callback are optional and work as described above.

```

const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers()
  .configure(socketio(3031));

app.listen(3030);

```

params.provider

For any [service method call](#) made through Socket.io `params.provider` will be set to `socketio`. In a [hook](#) this can for example be used to prevent external users from making a service method call:

```

app.service('users').hooks({
  before: [
    remove(hook) {

```

```
// check for if(hook.params.provider) to prevent any external call
if(hook.params.provider === 'socketio') {
    throw new Error('You can not delete a user via Socket.io');
}
});

});
```

params.query

`params.query` will contain the query parameters sent from the client.

Important: Only `params.query` is passed between the server and the client, other parts of `params` are not. This is for security reasons so that a client can't set things like `params.user` or the database options. You can always map from `params.query` to `params` in a before hook.

uWebSocket

The options can also be used to initialize [uWebSocket](#) which is a WebSocket server implementation that provides better performance and reduced latency.

```
$ npm install uws --save
```

```
const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers();

app.configure(socketio({
  wsEngine: 'uws'
}));

app.listen(3030);
```

Middleware and service parameters

[Socket.io middleware](#) can modify the `feathers` property on the `socket` which will then be used as the service parameters:

```
app.configure(socketio(function(io) {
  io.use(function (socket, next) {
    socket.feathers.user = { name: 'David' };
    next();
  });
}));

app.use('messages', {
  create(data, params, callback) {
    // When called via SocketIO:
    params.provider // -> socketio
    params.user // -> { name: 'David' }
  }
});
```

Client

The `client` module in `feathers-socketio` (`require('feathers-socketio/client')`) allows to connect to services exposed through the [Socket.io server](#) via a Socket.io socket.

Very important: The examples below assume you are using Feathers either in Node or in the browser with a module loader like Webpack or Browserify. For using Feathers with a `<script>` tag, AMD modules or with React Native see the [client chapter](#).

Note: A client application can only use a single transport (either REST, Socket.io or Primus). Using two transports in the same client application is normally not necessary.

socketio(socket)

Initialize the Socket.io client using a given socket and the default options.

```
const feathers = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');

const socket = io('http://api.feathersjs.com');
const app = feathers();

// Set up Socket.io client with the socket
app.configure(socketio(socket));

// Receive real-time events through Socket.io
app.service('messages')
  .on('created', message => console.log('New message created', message));

// Call the `messages` service
app.service('messages').create({
  text: 'A message from a REST client'
});
```

socketio(socket, options)

Initialize the Socket.io client using a given socket and the given options.

Options can be:

- `timeout` (default: 5000ms) - The time after which a method call fails and times out. This usually happens when calling a service or service method that does not exist.

```
const feathers = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');

const socket = io('http://api.feathersjs.com');
const app = feathers();

// Set up Socket.io client with the socket
// And a timeout of 2 seconds
app.configure(socketio(socket, {
  timeout: 2000
}));
```

Changing the socket client timeout

Currently, the only way for clients to determine if a service or service method exists is through a timeout. You can set the timeout either through the option above or on a per-service level by setting the `timeout` property:

```
app.service('messages').timeout = 3000;
```

Direct connection

Feathers sets up a normal Socket.io server that you can connect to with any Socket.io compatible client, usually the [Socket.io client](#) either by loading the `socket.io-client` module or `/socket.io/socket.io.js` from the server. Unlike HTTP calls, websockets do not have an inherent cross-origin restriction in the browser so it is possible to connect to any Feathers server.

ProTip: The socket connection URL has to point to the server root which is where Feathers will set up Socket.io.

```
<!-- Connecting to the same URL -->
<script src="/socket.io/socket.io.js">
<script>
  var socket = io();
</script>

<!-- Connecting to a different server -->
<script src="http://localhost:3030/socket.io/socket.io.js">
<script>
  var socket = io('http://localhost:3030/');
</script>
```

Calling service methods

Service methods can be called by emitting a `<servicepath>::<methodname>` event with the method parameters. `servicepath` is the name the service has been registered with (in `app.use`) without leading or trailing slashes. An optional callback following the `function(error, data)` Node convention will be called with the result of the method call or any errors that might have occurred.

`params` will be set as `params.query` in the service method call. Other service parameters can be set through a [Socket.io middleware](#).

find

Retrieves a list of all matching resources from the service

```
socket.emit('messages::find', { status: 'read', user: 10 }, (error, data) => {
  console.log('Found all messages', data);
});
```

Will call `messages.find({ query: { status: 'read', user: 10 } })` on the server.

get

Retrieve a single resource from the service.

```
socket.emit('messages::get', 1, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, {})` on the server.

```
socket.emit('messages::get', 1, { fetch: 'all' }, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
socket.emit('messages::create', {
  "text": "I really have to iron"
}, (error, message) => {
  console.log('Todo created', message);
});
```

Will call `messages.create({ "text": "I really have to iron" }, {})` on the server.

```
socket.emit('messages::create', [
  { "text": "I really have to iron" },
  { "text": "Do laundry" }
]);
```

Will call `messages.create` with the array.

update

Completely replace a single or multiple resources.

```
socket.emit('messages::update', 2, {
  "text": "I really have to do laundry"
}, (error, message) => {
  console.log('Todo updated', message);
});
```

Will call `messages.update(2, { "text": "I really have to do laundry" }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
socket.emit('messages::update', null, {
  complete: true
}, { complete: false });
```

Will call `messages.update(null, { "complete": true }, { query: { complete: 'false' } })` on the server.

ProTip: `update` is normally expected to replace an entire resource which is why the database adapters only support `patch` for multiple records.

patch

Merge the existing data of a single or multiple resources with the new `data`.

```
socket.emit('messages::patch', 2, {
  read: true
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `messages.patch(2, { "read": true }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
socket.emit('messages::patch', null, {
  complete: true
}, {
  complete: false
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `messages.patch(null, { complete: true }, { query: { complete: false } })` on the server to change the status for all read messages.

This is supported out of the box by the Feathers [database adapters](#)

remove

Remove a single or multiple resources:

```
socket.emit('messages::remove', 2, { cascade: true }, (error, message) => {
  console.log('Removed a message', message);
});
```

Will call `messages.remove(2, { query: { cascade: true } })` on the server. The `id` can also be `null` to remove multiple resources:

```
socket.emit('messages::remove', null, { read: true });
```

Will call `messages.remove(null, { query: { read: 'true' } })` on the server to delete all read messages.

Listening to events

Listening to service events allows real-time behaviour in an application. [Service events](#) are sent to the socket in the form of `servicepath eventname`.

created

The `created` event will be published with the callback data when a service `create` returns successfully.

```
var socket = io('http://localhost:3030/');

socket.on('messages created', function(message) {
  console.log('Got a new Todo!', message);
});
```

updated, patched

The `updated` and `patched` events will be published with the callback data when a service `update` or `patch` method calls back successfully.

```
var socket = io('http://localhost:3030/');

socket.on('my/messages updated', function(message) {
  console.log('Got an updated Todo!', message);
});
```

```
socket.emit('my/messages::update', 1, {
  text: 'Updated text'
}, {}, function(error, callback) {
  // Do something here
});
```

removed

The `removed` event will be published with the callback data when a service `remove` calls back successfully.

```
var socket = io('http://localhost:3030/');

socket.on('messages removed', function(message) {
  // Remove element showing the Todo from the page
  $('#message-' + message.id).remove();
});
```

Primus

[Star](#) 14 npm v2.2.0 [changelog .md](#)

```
$ npm install feathers-primus --save
```

The [feathers-primus](#) module allows to call [service methods](#) and receive [real-time events](#) via [Primus](#), a universal wrapper for real-time frameworks that supports Engine.IO, WebSockets, Faye, BrowserChannel, SockJS and Socket.IO.

Service method	Method event name	Real-time event
.find()	messages::find	-
.get()	messages::get	-
.create()	messages::create	messages created
.update()	messages::update	messages updated
.patch()	messages::patch	messages patched
.remove()	messages::removed	messages removed

Important: Primus is also used to *call* service methods. Using sockets for both, calling methods and receiving real-time events is generally faster than using [REST](#) and there is usually no need to use both, REST and Socket.io in the same client application at the same time.

Server

Additionally to `feathers-primus` your websocket library of choice also has to be installed.

```
$ npm install ws --save
```

`app.configure(primus(options [, callback]))`

Sets up the Primus transport with the given [Primus options](#) and optionally calls the callback with the Primus server instance.

Pro tip: Once the server has been started with `app.listen()` or `app.setup(server)` the Primus server object is available as `app.primus`.

```
const feathers = require('feathers');
const primus = require('feathers-primus');

const app = feathers();

// Set up Primus with SockJS
app.configure(feathers.primus({
  transformer: 'sockjs'
}, function(primus) {
  // Do something with primus object
}));
```

params.provider

For any [service method call](#) made through `params.provider` will be set to `primus`. In a [hook](#) this can for example be used to prevent external users from making a service method call:

```
app.service('users').hooks({
  before: [
    remove(hook) {
      // check for if(hook.params.provider) to prevent any external call
      if(hook.params.provider === 'primus') {
        throw new Error('You can not delete a user via Primus');
      }
    }
  ],
});
```

params.query

`params.query` will contain the query parameters sent from the client.

Important: Only `params.query` is passed between the server and the client, other parts of `params` are not. This is for security reasons so that a client can't set things like `params.user` or the database options. You can always map from `params.query` to `params` in a [before hook](#).

Middleware and service parameters

The Primus request object has a `feathers` property that can be extended with additional service `params` during authorization:

```
app.configure(primus({
  transformer: 'sockjs'
}, function(primus) {
  // Do something with primus
  primus.use('todos::create', function(socket, done){
    // Exposing a request property to services and hooks
    socket.request.feathers.referrer = socket.request.referrer;
    done();
  });
}));

app.use('messages', {
  create(data, params, callback) {
    // When called via Primus:
    params.provider // -> primus
    params.user // -> { name: 'David' }
  }
});
```

Client

The `client` module in `feathers-primus` (`require('feathers-primus/client')`) allows to connect to services exposed through the [Primus server](#) via a client socket.

Very important: The examples below assume you are using Feathers either in Node or in the browser with a module loader like Webpack or Browserify. For using Feathers with a `<script>` tag, AMD modules or with React Native see the [client chapter](#).

Note: A client application can only use a single transport (either REST, Socket.io or Primus). Using two transports in the same client application is normally not necessary.

Loading the Primus client library

In the browser the Primus client library (usually at `primus/primus.js`) always has to be loaded using a `<script>` tag:

```
<script type="text/javascript" src="primus/primus.js"></script>
```

Important: This will make the `Primus` object globally available. Module loader options are currently not available.

Client use in NodeJS

In NodeJS a Primus client can be initialized as follows:

```
const Primus = require('primus');
const Emitter = require('primus-emitter');
const Socket = Primus.createSocket({
  transformer: 'websockets',
  plugin: {
    'emitter': Emitter
  }
});
const socket = new Socket('http://api.feathersjs.com');
```

primus(socket)

Initialize the Socket.io client using a given socket and the default options.

```
const feathers = require('feathers');
const primus = require('feathers-primus/client');
const socket = new Primus('http://api.my-feathers-server.com');

const app = feathers();

app.configure(primus(socket));

// Receive real-time events through Socket.io
app.service('messages')
  .on('created', message => console.log('New message created', message));

// Call the `messages` service
app.service('messages').create({
  text: 'A message from a REST client'
});
```

primus(socket, options)

Initialize the Socket.io client using a given socket and the given options.

Options can be:

- `timeout` (default: 5000ms) - The time after which a method call fails and times out. This usually happens when calling a service or service method that does not exist.

```
const feathers = require('feathers');
const primus = require('feathers-primus/client');
```

```
const socket = new Primus('http://api.my-feathers-server.com');

const app = feathers();

app.configure(primus(socket, { timeout: 2000 }));
```

Changing the socket client timeout

Currently, the only way for clients to determine if a service or service method exists is through a timeout. You can set the timeout either through the option above or on a per-service level by setting the `timeout` property:

```
app.service('messages').timeout = 3000;
```

Direct connection

In the browser, the connection can be established by loading the client from `primus/primus.js` and instantiating a new `Primus` instance. Unlike HTTP calls, websockets do not have a cross-origin restriction in the browser so it is possible to connect to any Feathers server.

See the [Primus docs](#) for more details.

ProTip: The socket connection URL has to point to the server root which is where Feathers will set up Primus.

```
<script src="primus/primus.js">
<script>
  var socket = new Primus('http://api.my-feathers-server.com');
</script>
```

Calling service methods

Service methods can be called by emitting a `<servicepath>::<methodname>` event with the method parameters. `servicepath` is the name the service has been registered with (in `app.use`) without leading or trailing slashes. An optional callback following the `function(error, data)` Node convention will be called with the result of the method call or any errors that might have occurred.

`params` will be set as `params.query` in the service method call. Other service parameters can be set through a [Primus middleware](#).

find

Retrieves a list of all matching resources from the service

```
primus.send('messages::find', { status: 'read', user: 10 }, (error, data) => {
  console.log('Found all messages', data);
});
```

Will call `messages.find({ query: { status: 'read', user: 10 } })` on the server.

get

Retrieve a single resource from the service.

```
primus.send('messages::get', 1, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, {})` on the server.

```
primus.send('messages::get', 1, { fetch: 'all' }, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
primus.send('messages::create', {
  "text": "I really have to iron"
}, (error, message) => {
  console.log('Message created', message);
});
```

Will call `messages.create({ "text": "I really have to iron" }, {})` on the server.

```
primus.send('messages::create', [
  { "text": "I really have to iron" },
  { "text": "Do laundry" }
]);
```

Will call `messages.create` on the server with the array.

update

Completely replace a single or multiple resources.

```
primus.send('messages::update', 2, {
  "text": "I really have to do laundry"
}, (error, message) => {
  console.log('Message updated', message);
});
```

Will call `messages.update(2, { "text": "I really have to do laundry" }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
primus.send('messages::update', null, {
  complete: true
}, { complete: false });
```

Will call `messages.update(null, { "complete": true }, { query: { complete: 'false' } })` on the server.

ProTip: `update` is normally expected to replace an entire resource which is why the database adapters only support `patch` for multiple records.

patch

Merge the existing data of a single or multiple resources with the new `data`.

```
primus.send('messages::patch', 2, {
  read: true
```

```

}, (error, message) => {
  console.log('Patched message', message);
});

```

Will call `messages.patch(2, { "read": true }, {})` on the server. The `id` can also be `null` to update multiple resources:

```

primus.send('messages::patch', null, {
  complete: true
}, {
  complete: false
}, (error, message) => {
  console.log('Patched message', message);
});

```

Will call `messages.patch(null, { complete: true }, { query: { complete: false } })` on the server to change the status for all read messages.

This is supported out of the box by the Feathers [database adapters](#)

remove

Remove a single or multiple resources:

```

primus.send('messages::remove', 2, { cascade: true }, (error, message) => {
  console.log('Removed a message', message);
});

```

Will call `messages.remove(2, { query: { cascade: true } })` on the server. The `id` can also be `null` to remove multiple resources:

```
primus.send('messages::remove', null, { read: true });
```

Will call `messages.remove(null, { query: { read: 'true' } })` on the server to delete all read messages.

Listening to events

Listening to service events allows real-time behaviour in an application. [Service events](#) are sent to the socket in the form of `servicepath eventname`.

created

The `created` event will be published with the callback data when a service `create` returns successfully.

```

primus.on('messages created', function(message) {
  console.log('Got a new Message!', message);
});

```

updated, patched

The `updated` and `patched` events will be published with the callback data when a service `update` or `patch` method calls back successfully.

```

primus.on('my/messages updated', function(message) {
  console.log('Got an updated Message!', message);
});

```

```
});

primus.send('my/messages::update', 1, {
  text: 'Updated text'
}, {}, function(error, callback) {
  // Do something here
});
```

removed

The `removed` event will be published with the callback data when a service `remove` calls back successfully.

```
primus.on('messages removed', function(message) {
  // Remove element showing the Message from the page
  $('#message-' + message.id).remove();
});
```

Authentication

 Star 210 npm v1.2.7 changelog .md

```
$ npm install feathers-authentication --save
```

The [feathers-authentication](#) module assists in using JWT for authentication. It has three primary purposes:

1. Setup an `/authentication` endpoint to create JSON Web Tokens (JWT). JWT are used as access tokens. (learn more about JWT at [jwt.io](#))
2. Provide a consistent authentication API for all of the Feathers transports: `feathers-rest`, `feathers-socketio`, and `feathers-primus`.
3. Provide a framework for authentication plugins that use [Passport](#) strategies to protect endpoints.

Complementary Plugins

The following plugins are complementary, but entirely optional:

- `feathers-authentication-client`
- `feathers-authentication-local`
- `feathers-authentication-jwt`
- `feathers-authentication-oauth1`
- `feathers-authentication-oauth2`

For the auth middleware to work as expected, the plugins must be configured before creating any services.

API

This module contains:

1. The main entry function
2. The `/authentication` service
3. The `authenticate` hook
4. Authentication Events
5. Express middleware
6. A [Passport](#) adapter for Feathers

Configuration

```
app.configure(auth(options))
```

Setup is done the same as all Feathers plugins, using the `configure` method:

```
const auth = require('feathers-authentication');

// Available options are listed in the "Default Options" section
app.configure(auth(options))
```

Default options

The following default options will be mixed in with your global `auth` object from your config file. It will set the mixed options back on to the app so that they are available at any time by calling `app.get('auth')`. They can all be overridden and are required by some of the authentication plugins.

```
{
  path: '/authentication', // the authentication service path
  header: 'Authorization', // the header to use when using JWT auth
  entity: 'user', // the entity that will be added to the request, socket, and hook.params. (ie. req.user, socket.user, hook.params.user)
  service: 'users', // the service to look up the entity
  passReqToCallback: true, // whether the request object should be passed to the strategies `verify` function
  session: false, // whether to use sessions
  cookie: {
    enabled: false, // whether cookie creation is enabled
    name: 'feathers-jwt', // the cookie name
    httpOnly: false, // when enabled, prevents the client from reading the cookie.
    secure: true // whether cookies should only be available over HTTPS
  },
  jwt: {
    header: { type: 'access' }, // by default is an access token but can be any type
    audience: 'https://yourdomain.com', // The resource server where the token is processed
    subject: 'anonymous', // Typically the entity id associated with the JWT
    issuer: 'feathers', // The issuing server, application or resource
    algorithm: 'HS256', // the algorithm to use
    expiresIn: '1d' // the access token expiry
  }
}
```

Additional app methods

The Feathers `app` will contain two useful methods once you've configured the auth plugin:

- `app.passport.createJWT`
- `app.passport.verifyJWT`

`app.passport.createJWT(payload, options) => promise source`

This is the method used by the `/authentication` service to generate JSON Web Tokens.

- `payload {Object}` - becomes the JWT payload. Will also include an `exp` property denoting the expiry timestamp.
- `options {Object}` - the options passed to `jsonwebtoken sign()`
 - `secret {String | Buffer}` - either the secret for HMAC algorithms, or the PEM encoded private key for RSA and ECDSA.
 - `jwt` - See the `jsonwebtoken` package docs for other available options. The authenticate method uses the default `jwt` options. When using this package directly, they will have to be passed in manually.

The returned `promise` resolves with the JWT or fails with an error.

`app.passport.verifyJWT(token, options) source`

Verifies the signature and payload of the passed in JWT `token` using the `options`.

- `token {JWT}` - the JWT to be verified.
- `options {Object}` the options passed to `jsonwebtoken verify()`

- o `secret {String | Buffer}` -- either the secret for HMAC algorithms, or the PEM encoded private key for RSA and ECDSA.
- o See the [jsonwebtoken](#) package docs for other available options.

The authentication service

The heart of this plugin is simply a service for creating JWT. It's a normal Feathers service that implements only the `create` and `remove` methods. The `/authentication` service provides all of the functionality that the `/auth/local` and `/auth/token` endpoints did. To choose a strategy, the client must pass the `strategy` name in the request body. This will be different based on the plugin used. See the documentation for the plugins listed at the top of this page for more information.

`app.service('/authentication').create(data, params)`

The `create` method will be used in nearly every Feathers application. It creates a JWT based on the `jwt` options configured on the plugin. The API of this method utilizes the `hook` object:

before hook API:

These properties can be modified to change the behavior of the `/authentication` service.

- `hook.data.payload {Object}` - determines the payload of the JWT
- `hook.params.payload {Object}` - also determines the payload of the JWT. Any matching attributes in the `hook.data.payload` will be overwritten by these. Persists into after hooks.
- `hook.params.authenticated {Boolean}` - After successful authentication, will be set to `true`, unless it's set to `false` in a before hook. If you set it to `false` in a before hook, it will prevent the websocket from being flagged as authenticated. Persists into after hooks.

after hook API:

- `hook.params[entity] {Object}` - After successful authentication, the `entity` looked up from the database will be populated here. (The default option is `user`.)

`app.service('/authentication').remove(data)`

The `remove` method will be used less often. It mostly exists to allow for adding hooks the the "logout" process. For example, in services that require high control over security, a developer could register hooks on the `remove` method that perform token blacklisting.

after hook API:

- `hook.result {Object}` - After logout, useful information regarding the previous session will be populated here.

Below is the example of the hook usage:

```
after: {
  remove: [
    function (hook) {
      return app.passport.verifyJWT(hook.result.accessToken, { secret: app.passport.options('jwt').secret })
    }
    .then((data) => {
      // removing the user who decided to logout
      app.service('users').remove(data.userId).then(() => {
        return hook;
      })
    })
  ]
}
```

```

        });
    });
}
]
```

The `authenticate` hook

`auth.hooks.authenticate(strategies)`, where `strategies` is an array of passport strategy names.

`feathers-authentication` only includes a single hook. This bundled `authenticate` hook is used to register an array of authentication strategies on a service method. When authenticating, the client must send the `strategy` as part of the payload.

Note: This should usually be used on your `/authentication` service. Without it you can hit the `authentication` service and generate a JWT `accessToken` without authentication (ie. anonymous authentication).

```

app.service('authentication').hooks({
  before: {
    create: [
      // You can chain multiple strategies
      auth.hooks.authenticate(['jwt', 'local']),
    ],
    remove: [
      auth.hooks.authenticate('jwt')
    ]
  }
});
```

The hooks that were once bundled with this module are now located at [feathers-authentication-hooks](#).

Authentication Events

The `login` and `logout` events are emitted on the `app` object whenever a client successfully authenticates or "logs out". (With JWT, logging out doesn't invalidate the JWT. Read the section about how JWT work for more information.) These events are only emitted on the server.

`app.on('login', callback))` and `app.on('logout', callback))`

These two events use a callback with the same signature.

- `callback {Function}` - a function in the format `function (result, meta) {}`.
 - `result {Object}` - The final `hook.result` from the `authentication` service. Unless you customize the `hook.response` in an after hook, this will only contain the `accessToken`, which is the JWT.
 - `meta {Object}` - information about the request. *The `meta` data varies per transport/provider as follows.*
 - Using `feathers-rest`
 - `provider {String}` - will always be "rest"
 - `req {Object}` - the Express request object.
 - `res {Object}` - the Express response object.
 - Using `feathers-socketio` and `feathers-primus`:
 - `provider {String}` - the transport name: `socketio` OR `primus`
 - `connection {Object}` - the same as `params` in the hook context
 - `socket {SocketObject}` - the current user's WebSocket object. It also contains the `feathers` attribute, which is the same as `params` in the hook context.

Express Middleware

There is an `authenticate` middleware. It is used the exact same way you would the regular Passport express middleware:

```
app.post('/login', auth.express.authenticate('local', { successRedirect: '/app', failureRedirect: '/login' }));
```

Additional middleware are included and exposed but typically you don't need to worry about them:

- `emitEvents` `source` - emit `login` and `logout` events
- `exposeCookies` `source` - expose cookies to Feathers so they are available to hooks and services. **This is NOT used by default as its use exposes your API to CSRF vulnerabilities.** Only use it if you really know what you're doing.
- `exposeHeaders` `source` - expose headers to Feathers so they are available to hooks and services. **This is NOT used by default as its use exposes your API to CSRF vulnerabilities.** Only use it if you really know what you're doing.
- `failureRedirect` `source` - support redirecting on auth failure. Only triggered if `hook.redirect` is set.
- `successRedirect` `source` - support redirecting on auth success. Only triggered if `hook.redirect` is set.
- `setCookie` `source` - support setting the JWT access token in a cookie. Only enabled if cookies are enabled.

Note: Feathers will NOT read an access token from a cookie. This would expose the API to CSRF attacks.

This `setCookie` feature is available primarily for helping with Server Side Rendering.

Migrating to 1.x

Refer to [the migration guide](#).

Complete Example

Here's an example of a Feathers server that uses `feathers-authentication` for local auth. You can try it out on your own machine by running the [example](#)

For the auth middleware to work as expected, the plugins must be configured before creating any services.

```
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const hooks = require('feathers-hooks');
const memory = require('feathers-memory');
const bodyParser = require('body-parser');
const errors = require('feathers-errors');
const errorHandler = require('feathers-errors/handler');
const local = require('feathers-authentication-local');
const jwt = require('feathers-authentication-jwt');
const auth = require('feathers-authentication');

const app = feathers();
app.configure(rest())
  .configure(socketio())
  .configure(hooks())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  .configure(auth({ secret: 'supersecret' }))
  .configure(local())
  .configure(jwt())
  .use('/users', memory())
  .use('/', feathers.static(__dirname + '/public'))
```

```
.use(errorHandler());

app.service('authentication').hooks({
  before: [
    create: [
      // You can chain multiple strategies
      auth.hooks.authenticate(['jwt', 'local'])
    ],
    remove: [
      auth.hooks.authenticate('jwt')
    ]
  }
});

// Add a hook to the user service that automatically replaces
// the password with a hash of the password before saving it.
app.service('users').hooks({
  before: [
    find: [
      auth.hooks.authenticate('jwt')
    ],
    create: [
      local.hooks.hashPassword({ passwordField: 'password' })
    ]
  }
});

const port = 3030;
let server = app.listen(port);
server.on('listening', function() {
  console.log(`Feathers application started on localhost:${port}`);
});
```

Authentication Client



```
npm install feathers-authentication-client --save
```

Note: This is only compatible with `feathers-authentication@1.x` and above.

The `feathers-authentication-client` module allows you to easily authenticate against a Feathers server. It is not required. It simply makes it easier to implement authentication in your client by automatically storing and sending the JWT access token and handling re-authenticating when a websocket disconnects.

API

This module contains:

- [The main entry function](#)
- [Additional feathersClient methods](#)
- [Some helpful hooks](#)

Configuration

`feathersClient.configure(auth(options))`

Setup is done the same as all Feathers plugins, using the `configure` method:

```
import auth from 'feathers-authentication-client';

// Available options are listed in the "Default Options" section
feathersClient.configure(auth(options))
```

The `transports plugins` must have been initialized previously to the authentication plugin on the client side

Default options

The following default options will be mixed in with the settings you pass in when configuring authentication. It will set the mixed options back to the app so that they are available at any time by `app.get('auth')`. They can all be overridden.

```
{
  header: 'Authorization', // the default authorization header for REST
  path: '/authentication', // the server-side authentication service path
  jwtStrategy: 'jwt', // the name of the JWT authentication strategy
  entity: 'user', // the entity you are authenticating (ie. a users)
  service: 'users', // the service to look up the entity
  cookie: 'feathers-jwt', // the name of the cookie to parse the JWT from when cookies are enabled server side
  storageKey: 'feathers-jwt', // the key to store the accessToken in localstorage or AsyncStorage on React Native
  storage: undefined // Passing a WebStorage-compatible object to enable automatic storage on the client.
}
```

To enable `localStorage` on the client, be sure to set `storage: window.localStorage` in the client options. You can also provide other WebStorage-compatible objects. Here are a couple of useful storage packages:

- [localForage](#) helps deal with older browsers and browsers in Incognito / Private Browsing mode.
- [cookie-storage](#) uses cookies. It can be useful devices that don't support `localStorage`.

Additional feathersClient methods

After configuring this plugin, the Feathers client will have a few additional methods:

feathersClient.authenticate(options) [source](#)

Authenticate with a Feathers server by passing a `strategy` and other properties as credentials. It will use whichever transport has been setup on the client (`feathers-rest`, `feathers-socketio`, or `feathers-primus`). Returns a Promise.

```
feathersClient.authenticate({
  strategy: 'jwt',
  accessToken: '<the.jwt.token.string>'
})
```

- `data {Object}` - of the format `{strategy [, ...otherProps]}`
 - `strategy {String}` - the name of the strategy to be used to authenticate. Required.
 - `...otherProps {Properties}` vary depending on the chosen strategy. Above is an example of using the `jwt` strategy. Below is one for the `local` strategy.

```
feathersClient.authenticate({
  strategy: 'local',
  email: 'my@email.com',
  password: 'my-password'
})
```

When using `feathers-socketio` or `feathers-primus`, the WebSocket connection has to be authenticated by calling `app.authenticate()` in order to make requests using a stored `accessToken`.

```
app.authenticate().then(response => {
  /* make authenticated requests here */
  return response
})
```

feathersClient.logout() [source](#)

Removes the JWT `accessToken` from storage on the client. It also calls the `remove` method of the `/authentication` service on the Feathers server.

feathersClient.passport.getJWT() [source](#)

Pull the JWT from localstorage or the cookie. Returns a Promise.

feathersClient.passport.verifyJWT(token) [source](#)

Verify that a JWT is not expired and decode it to get the payload. Returns a Promise.

feathersClient.passport.payloadIsValid(token) [source](#)

Synchronously verify that a token has not expired. Returns a Boolean.

Hooks

There are 3 hooks. They are really meant for internal use and you shouldn't need to worry about them very often.

- `populateAccessToken` - Takes the token and puts it in `hooks.params.accessToken` in case you need it in one of your client side services or hooks
- `populateHeader` - Add the `accessToken` to the authorization header
- `populateEntity` - Experimental. Populate an entity based on the JWT payload.

Complete Example

Here's an example of a Feathers server that uses `feathers-authentication-client`.

```
const feathers = require('feathers/client');
const rest = require('feathers-rest/client');
const superagent = require('superagent');
const hooks = require('feathers-hooks');
const localStorage = require('localStorage-memory');
const auth = require('feathers-authentication-client');

const feathersClient = feathers();

feathersClient.configure(hooks())
  .configure(rest('http://localhost:3030').superagent(superagent))
  .configure(auth({ storage: localStorage }));

feathersClient.authenticate({
  strategy: 'local',
  email: 'admin@feathersjs.com',
  password: 'admin'
})
.then(response => {
  console.log('Authenticated!', response);
  return feathersClient.passport.verifyJWT(response.accessToken);
})
.then(payload => {
  console.log('JWT Payload', payload);
  return feathersClient.service('users').get(payload.userId);
})
.then(user => {
  feathersClient.set('user', user);
  console.log('User', feathersClient.get('user'));
})
.catch(function(error){
  console.error('Error authenticating!', error);
});
```

Handling the special re-authentication errors

In the event that your server goes down or the client loses connectivity, it will automatically handle attempting to re-authenticate the socket when the client regains connectivity with the server. In order to handle an authentication failure during automatic re-authentication you need to implement the following event listener:

```
const errorHandler = error => {
  app.authenticate({
    strategy: 'local',
    email: 'admin@feathersjs.com',
```

```
    password: 'admin'  
}).then(response => {  
    // You are now authenticated again  
});  
};  
  
// Handle when auth fails during a reconnect or a transport upgrade  
app.on('reauthentication-error', errorHandler)
```

Local Authentication



```
$ npm install feathers-authentication-local --save
```

[feathers-authentication-local](#) is a server side module that wraps the [passport-local](#) authentication strategy, which lets you authenticate with your Feathers application using a username and password.

This module contains 3 core pieces:

1. The main initialization function
2. The `hashPassword` hook
3. The `Verifier` class

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('feathers');
const authentication = require('feathers-authentication');
const local = require('feathers-authentication-local');
const app = feathers();

// Setup authentication
app.configure(authentication(settings));
app.configure(local());

// Setup a hook to only allow valid JWTs or successful
// local auth to authenticate and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['local', 'jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Default Options

```
{
  name: 'local', // the name to use when invoking the authentication Strategy
  entity: 'user', // the entity that you're comparing username/password against
  service: 'users', // the service to look up the entity
  usernameField: 'email', // key name of username field
  passwordField: 'password', // key name of password field
  passReqToCallback: true, // whether the request object should be passed to `verify`
  session: false // whether to use sessions,
  Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for
  details.
}
```

hashPassword hook

This hook is used to hash plain text passwords before they are saved to the database. It uses the bcrypt algorithm by default but can be customized by passing your own `options.hash` function.

```
const local = require('feathers-authentication-local');

app.service('users').hooks({
  before: {
    create: [
      local.hooks.hashPassword()
    ]
  }
});
```

Default Options

```
{
  passwordField: 'password', // key name of password field to look on hook.data
  hash: customHashFunction // default is the bcrypt hash function. Takes in a password and returns a hash.
}
```

Verifier

This is the verification class that does the actual username and password verification by looking up the entity (normally a `user`) on a given service by the `usernameField` and compares the hashed password using bcrypt. It has the following methods that can all be overridden. All methods return a promise except `verify`, which has the exact same signature as [passport-local](#).

```
{
  constructor(app, options) // the class constructor
  _comparePassword(entity, password) // compares password using bcrypt
  _normalizeResult(result) // normalizes result from service to account for pagination
  verify(req, username, password, done) // queries the service and calls the other internal functions.
}
```

Customizing the Verifier

The `Verifier` class can be extended so that you customize its behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want to use this plugin.

An example of customizing the Verifier:

```
import local, { Verifier } from 'feathers-authentication-local';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, username, password, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}
```

```
app.configure(local({ Verifier: CustomVerifier }));
```

Client Usage

When this module is registered server side, using the default config values this is how you can authenticate using `feathers-authentication-client`:

```
app.authenticate({
  strategy: 'local',
  email: 'your email',
  password: 'your password'
}).then(response => {
  // You are now authenticated
});
```

Direct Usage

Using a HTTP Request

If you are not using the `feathers-authentication-client` and you have registered this module server side then you can simply make a `POST` request to `/authentication` with the following payload:

```
// POST /authentication the Content-Type header set to application/json
{
  "strategy": "local",
  "email": "your email",
  "password": "your password"
}
```

Here is what that looks like with curl:

```
curl -H "Content-Type: application/json" -X POST -d '{"strategy":"local","email":"your email","password":"your password"}' http://localhost:3030/authentication
```

Using Sockets

Authenticating using a local strategy via sockets is done by emitting the following message:

```
const io = require('socket.io-client');
const socket = io('http://localhost:3030');

socket.emit('authenticate', {
  strategy: 'local',
  email: 'your email',
  password: 'your password'
}, function(message, data) {
  console.log(message); // message will be null
  console.log(data); // data will be {"accessToken": "your token"}
  // You can now send authenticated messages to the server
});
```


Local Authentication Management

```
$ npm install feathers-authentication-management --save
```

Sign up verification, forgotten password reset, and other capabilities for local authentication.

Multiple communication channels:

Traditionally users have been authenticated using their `username` or `email`. However that landscape is changing.

Teens are more involved with cellphone SMS, whatsapp, facebook, QQ and wechat than they are with email. Seniors may not know how to create an email account or check email, but they have smart phones and perhaps whatsapp or wechat accounts.

A more flexible design would maintain multiple communication channels for a user -- username, email address, phone number, handles for whatsapp, facebook, QQ, wechat -- which each uniquely identify the user. The user could then sign in using any of their unique identifiers. The user could also indicate how they prefer to be contacted. Some may prefer to get password resets via long tokens sent by email; others may prefer short numeric tokens sent by SMS or wechat.

`feathers-authentication` and `feathers-authentication-management` provide much of the infrastructure necessary to implement such a scenario.

Capabilities:

- Checking that values for fields like `username`, `email`, `cellphone` are unique within `users` items.
- Hooks for adding a new user.
- Send another sign up verification notification, routing through user's selected transport.
- Process a sign up or identity change verification from a URL response.
- Process a sign up or identity change verification using a short token.
- Send a forgotten password reset notification, routing through user's preferred communication transport.
- Process a forgotten password reset from a URL response.
- Process a forgotten password reset using a short token.
- Process password change.
- Process an identity change such as a new email addr, or cellphone.

User notifications may be sent for:

- Sign up verification when a new user is created.
- Resending a signup verification, e.g. previous verification was lost or is expired.
- Successful user verification.
- Resetting the password when the password is forgotten.
- Successful password reset for a forgotten password.
- Manual change of a password.
- Change of identity. Notify both the current and new e.g. old email addr may be notified when the email addr changes.

May be used with

- `feathers-client` service calls over websockets or HTTP.

- Client side wrappers for `feathers-client` service calls.
- HTTP POST calls.
- React's Redux.
- Vue (docs to do)

Various-sized tokens can be used during the verify/reset processes:

A 30-char token is generated suitable for URL responses. (Configurable length.) This may be embedded in URL links sent by email, SMS or social media so that clicking the link starts the sign up verification or the password reset.

A 6-digit token is also generated suitable for notification by SMS or social media. (Configurable length, may be alpha-numeric instead.) This may be manually entered in a UI to start the sign up verification or the password reset.

The email verification token has a 5-day expiry (configurable), while the password reset has a 2 hour expiry (configurable).

Typically your notifier routine refers to a property like `user.preferredComm: 'email'` to determine which transport to use for user notification. However the API allows the UI to be set up to ask the user which transport they prefer for that time.

The server does not handle any interactions with the user. Leaving it a pure API server, lets it be used with both native and browser clients.

Contents

- [The Service](#)
- [Client](#)
 - [Using Feathers' method calls](#)
 - [Provided service wrappers](#)
 - [React's redux](#)
 - [Dispatching services](#)
 - [Dispatching authentication](#)
- [Hooks](#)
- [Multiple services](#)
- [Database](#)
- [Routing](#)
- [Security](#)
- [Configurable](#)

Service

```
import authManagement from 'feathers-authentication-management';
app.configure(authentication)
  .configure(authManagement({ options }))
  .configure(user);
```

options are:

- `service`: The path of the service for user items, e.g. `/users` (default) or `/organization`.
- `path`: The path to associate with this service. Default `authManagement`. See [Multiple services](#) for more information.
- `notifier`: `function(type, user, notifierOptions)` returns a Promise.
 - `type`: type of notification
 - 'resendVerifySignup' From `resendVerifySignup` API call

- 'verifySignup' From verifySignupLong and verifySignupShort API calls
- 'sendResetPwd' From sendResetPwd API call
- 'resetPwd' From resetPwdLong and resetPwdShort API calls
- 'passwordChange' From passwordChange API call
- 'identityChange' From identityChange API call
- user: user's item, minus password.
- notifierOptions: notifierOptions option from resendVerifySignup and sendResetPwd API calls
- longTokenLen: Half the length of the long token. Default is 15, giving 30-char tokens.
- shortTokenLen: Length of short token. Default is 6.
- shortTokenDigits: Short token is digits if true, else alphanumeric. Default is true.
- delay: Duration for sign up email verification token in ms. Default is 5 days.
- resetDelay: Duration for password reset token in ms. Default is 2 hours.
- skipIsVerifiedCheck: Allow 'sendResetPwd' and 'resetPwd' for unverified users. Default is false.
- identifyUserProps: Prop names in `user` item which uniquely identify the user, e.g. `['username', 'email', 'cellphone']`. The default is `['email']`. The prop values must be strings. Only these props may be changed with verification by the service. At least one of these props must be provided whenever a short token is used, as the short token alone is too susceptible to brute force attack.

The service creates and maintains the following properties in the `user` item:

- isVerified: If the user's email addr has been verified (boolean)
- verifyToken: The 30-char token generated for email addr verification (string)
- verifyShortToken: The 6-digit token generated for cellphone addr verification (string)
- verifyExpires: When the email addr token expire (Date)
- verifyChanges New values to apply on verification to some identifyUserProps (string array)
- resetToken: The 30-char token generated for forgotten password reset (string)
- resetShortToken: The 6-digit token generated for forgotten password reset (string)
- resetExpires: When the forgotten password token expire (Date)

The following `user` item might also contain the following props:

- preferredComm The preferred way to notify the user. One of identifyUserProps.

The `users` service is expected to be already configured. Its `patch` method is used to update the password when needed, and this module hashes the password before it is passed to `patch`, therefore `patch` may *not* have a `auth.hashPassword()` hook.

The user must be signed in before being allowed to change their password or communication values. The service, for feathers-authenticate v1.x, requires hooks similar to:

```
const isAction = (...args) => hook => args.includes(hook.data.action);
app.service('authManagement').before({
  create: [
    hooks.iff(isAction('passwordChange', 'identityChange'), auth.hooks.authenticate('jwt')),
  ],
});
```

Client

The service may be called on the client using

- Using Feathers method calls
- Provided service wrappers
- HTTP fetch

- React's Redux
- Vue 2.0 (docs todo)

Using Feathers method calls

Method calls return a Promise.

```

import authManagementService from 'feathers-authentication-management';
app.configure(authManagementService(options))
const authManagement = app.service('authManagement');

// check props are unique in the users items
authManagement.create({ action: 'checkUnique',
  value: identifyUser, // e.g. {email, username}. Props with null or undefined are ignored.
  ownId, // excludes your current user from the search
  meta: { noErrMsg }, // if return an error.message if not unique
})
// ownId allows the "current" item to be ignored when checking if a field value is unique among users.
// noErrMsg determines if the returned error.message contains text. This may simplify your client side validation.
on.

// resend sign up verification notification
authManagement.create({ action: 'resendVerifySignup',
  value: identifyUser, // {email}, {token: verifyToken}
  notifierOptions: {}, // options passed to options.notifier, e.g. {preferredComm: 'cellphone'}
})

// sign up or identityChange verification with long token
authManagement.create({ action: 'verifySignupLong',
  value: verifyToken, // compares to .verifyToken
})

// sign up or identityChange verification with short token
authManagement.create({ action: 'verifySignupShort',
  value: {
    user, // identify user, e.g. {email: 'a@a.com'}. See options.identifyUserProps.
    token, // compares to .verifyShortToken
  }
})

// send forgotten password notification
authManagement.create({ action: 'sendResetPwd',
  value: identifyUser, // {email}, {token: verifyToken}
  notifierOptions, // options passed to options.notifier, e.g. {preferredComm: 'email'}
})

// forgotten password verification with long token
authManagement.create({ action: 'resetPwdLong',
  value: {
    token, // compares to .resetToken
    password, // new password
  },
})

// forgotten password verification with short token
authManagement.create({ action: 'resetPwdShort',
  value: {
    user: identifyUser, // identify user, e.g. {email: 'a@a.com'}. See options.identifyUserProps.
    token, // compares to .resetShortToken
    password, // new password
  },
})

// change password
authManagement.create({ action: 'passwordChange',
  value: {
    user: identifyUser, // identify user, e.g. {email: 'a@a.com'}. See options.identifyUserProps.
  }
})

```

```

        oldPassword, // old password for verification
        password, // new password
    },
})

// change communications
authManagement.create({ action: 'identityChange',
    value: {
        user: identifyUser, // identify user, e.g. {email: 'a@a.com'}. See options.identifyUserProps.
        password, // current password for verification
        changes, // {email: 'a@a.com'} or {email: 'a@a.com', cellphone: '+1-800-555-1212'}
    },
})

// Authenticate user and log on if user is verified.
var cbCalled = false;
app.authenticate({ type: 'local', email, password })
.then((result) => {
    const user = result.data;
    if (!user || !user.isVerified) {
        app.logout();
        cb(new Error(user ? 'User\'s email is not verified.' : 'No user returned.'));
        return;
    }
    cbCalled = true;
    cb(null, user);
})
.catch((err) => {
    if (!cbCalled) { cb(err); } // ignore throws from .then( cb(null, user) )
});

```

Provided service wrappers

The wrappers return a Promise.

```

<script src=".../feathers-authentication-management/lib/client.js"></script>
or
import AuthManagement from 'feathers-authentication-management/lib/client';
const app = feathers() ...
const authManagement = new AuthManagement(app);

// check props are unique in the users items
authManagement.checkUnique(identifyUser, ownId, ifErrMsg)

// resend sign up verification notification
authManagement.resendVerifySignup(identifyUser, notifierOptions)

// sign up or identityChange verification with long token
authManagement.verifySignupLong(verifyToken)

// sign up or identityChange verification with short token
authManagement.verifySignupShort(verifyShortToken, identifyUser)

// send forgotten password notification
authManagement.sendResetPwd(identifyUser, notifierOptions)

// forgotten password verification with long token
authManagement.resetPwdLong(resetToken, password)

// forgotten password verification with short token
authManagement.resetPwdShort(resetShortToken, identifyUser, password)

// change password
authManagement.passwordChange(oldPassword, password, identifyUser)

// change identity

```

```
authManagement.identityChange(password, changesIdentifyUser, identifyUser)

// Authenticate user and log on if user is verified. v0.x only.
authManagement.authenticate(email, password)
```

React Redux

See `feathers-redux` for information about state, etc.

Dispatching services

```
import feathers from 'feathers-client';
import reduxifyServices from 'feathers-reduxify-services';
const app = feathers().configure(feathers.socketio(socket)).configure(feathers.hooks());
const services = reduxifyServices(app, ['users', 'authManagement', ...]);
...
// hook up Redux reducers
export default combineReducers({
  users: services.users.reducer,
  authManagement: services.authManagement.reducer,
});
...

// email addr verification with long token
// Feathers is now 100% compatible with Redux. Use just like [Feathers method calls.](#methods)
store.dispatch(services.authManagement.create({ action: 'verifySignupLong',
  value: verifyToken,
}, {}));
)
```

Dispatching authentication

User must be verified to sign in. v0.x only.

```
const reduxifyAuthentication = require('feathers-reduxify-authentication');
const signin = reduxifyAuthentication(app, { isUserAuthorized: (user) => user.isVerified });

// Sign in with the JWT currently in localStorage
if (localStorage['feathers-jwt']) {
  store.dispatch(signin.authenticate()).catch(err => { ... });
}

// Sign in with credentials
store.dispatch(signin.authenticate({ type: 'local', email, password }))
  .then(() => { ... })
  .catch(err => { ... });
```

Hooks

The service does not itself handle creation of a new user account nor the sending of the initial sign up verification request. Instead hooks are provided for you to use with the `users` service `create` method. If you set a service path other than the default of `'authManagement'`, the custom path name must be passed into the hook.

`verifyHooks.addVerification(path = 'authManagement')`

```
const verifyHooks = require('feathers-authentication-management').hooks;
// users service
module.exports.before = {
```

```

create: [
  auth.hashPassword(),
  verifyHooks.addVerification() // adds .isVerified, .verifyExpires, .verifyToken, .verifyChanges to the incoming data
]
};

module.exports.after = {
  create: [
    hooks.remove('password'),
    aHookToEmailYourVerification(),
    verifyHooks.removeVerification() // removes verification/reset fields other than .isVerified from the outgoing response
  ]
};

```

verifyHooks.isVerified()

A hook is provided to ensure the user's email addr is verified:

```

const auth = require('feathers-authentication').hooks;
const verifyHooks = require('feathers-authentication-management').hooks;
export.before = {
  create: [
    auth.authenticate('jwt'),
    verifyHooks.isVerified(),
  ]
};

```

Multiple services

We have considered until now situations where authentication was based on a user item. feathers-authorization however allows users to sign in with group and organization credentials as well as user ones.

You can easily configure `feathers-authentication-management` to handle such situations. Please refer to `test/multiInstances.test.js`.

Database

The service adds the following optional properties to the user item. You should add them to your user model if your database uses models.

```
{
  isVerified: { type: Boolean },
  verifyToken: { type: String },
  verifyShortToken: { type: String },
  verifyExpires: { type: Date }, // or a long integer
  verifyChanges: // an object (key-value map), e.g. { field: "value" }
  resetToken: { type: String },
  resetShortToken: { type: String },
  resetExpires: { type: Date }, // or a long integer
}
```

Routing

The client handles all interactions with the user. Therefore the server must serve the client app when, for example, a URL link is followed for email addr verification. The client must do some routing based on the path in the link.

Assume you have sent the email link: <http://localhost:3030/socket/verify/12b827994bb59cacce47978567989e>

The server serves the client app on `/socket`:

```
// Express-like middleware provided by Feathersjs.
app.use('/', serveStatic(app.get('public')))
  .use('/socket', (req, res) => {
    res.sendFile(path.resolve(__dirname, '..', 'public', 'socket.html')); // serve the client
  })
}
```

The client then routes itself based on the URL. You will likely use your favorite client-side router, but a primitive routing would be:

```
const [leader, provider, action, slug] = window.location.pathname.split('/');

switch (action) {
  case 'verify':
    verifySignUp(slug);
    break;
  case 'reset':
    resetPassword(slug);
    break;
  default:
    // normal app startup
}
```

Security

- The user must be identified when the short token is used, making the short token less appealing as an attack vector.
- The long and short tokens are erased on successful verification and password reset attempts. New tokens must be acquired for another attempt.
- API params are verified to be strings. If the param is an object, the values of its props are verified to be strings.
- `options.identifyUserProps` restricts the prop names allowed in param objects.
- In order to protect sensitive data, you should set a hook that prevent `PATCH` or `PUT` calls on authentication-management related properties:

```
// in user service hook
before: [
  update: [
    iff(isProvider('external'), preventChanges(
      'isVerified',
      'verifyToken',
      'verifyShortToken',
      'verifyExpires',
      'verifyChanges',
      'resetToken',
      'resetShortToken',
      'resetExpires'
    )),
  ],
  patch: [
    iff(isProvider('external'), preventChanges(
      'isVerified',
      'verifyToken',
      'verifyShortToken',
      'verifyExpires',
      'verifyChanges',
      'resetToken',
      'resetShortToken',
      'resetExpires'
    ))
  ]
]
```

```
  )),
  ],
},
```

Configurable

The length of the "30-char" token is configurable. The length of the "6-digit" token is configurable. It may also be configured as alphanumeric.

JWT Authentication



```
$ npm install feathers-authentication-jwt --save
```

[feathers-authentication-jwt](#) is a server side module that wraps the [passport-jwt](#) authentication strategy, which lets you authenticate with your Feathers application using a JSON Web Token ([JWT](#)) access token.

This module contains 3 core pieces:

1. The main initialization function
2. The `Verifier` class
3. The `ExtractJwt` object from `passport-jwt`.

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('feathers');
const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const app = feathers();

// Setup authentication
app.configure(authentication(settings));
app.configure(jwt());

// Setup a hook to only allow valid JWTs to authenticate
// and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Default Options

```
{
  name: 'jwt', // the name to use when invoking the authentication Strategy
  entity: 'user', // the entity that you pull from if an 'id' is present in the payload
  service: 'users', // the service to look up the entity
  passReqToCallback: true, // whether the request object should be passed to `verify`
  jwtFromRequest: [ // a passport-jwt option determining where to parse the JWT
    ExtractJwt.fromHeader, // From "Authorization" header
    ExtractJwt.fromAuthHeaderWithScheme('Bearer'), // Allowing "Bearer" prefix
    ExtractJwt.fromBodyField('body') // from request body
  ],
  secretOrKey: auth.secret, // Your main secret provided to passport-jwt
  session: false // whether to use sessions,
  Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for
```

```
    details.  
}
```

Additional [passport-jwt](#) options can be provided.

Verifier

This is the verification class that receives the JWT payload (if verification is successful) and either returns the payload or, if an `id` is present in the payload, populates the entity (normally a `user`) and returns both the entity and the payload. It has the following methods that can all be overridden. The `verify` function has the exact same signature as [passport-jwt](#).

```
{
  constructor(app, options) // the class constructor
  verify(req, payload, done) // queries the configured service
}
```

Customizing the Verifier

The `Verifier` class can be extended so that you customize its behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want use this plugin.

An example of customizing the Verifier:

```
import jwt, { Verifier } from 'feathers-authentication-jwt';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, payload, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}

app.configure(jwt({ verifier: CustomVerifier }));
```

Client Usage

When this module is registered server side, using the default config values this is how you can authenticate using `feathers-authentication-client`:

```
app.authenticate({
  strategy: 'jwt',
  accessToken: 'your access token'
}).then(response => {
  // You are now authenticated
});
```

Direct Usage

Using a HTTP Request

If you are not using the `feathers-authentication-client` and you have registered this module server side then you can simply include the access token in an `Authorization` header.

Here is what that looks like with curl:

```
curl -H "Content-Type: application/json" -H "Authorization: <your access token>" -X POST http://localhost:3030/authentication
```

Using Sockets

Authenticating using an access token via sockets is done by emitting the following message:

```
const io = require('socket.io-client');
const socket = io('http://localhost:3030');

socket.emit('authenticate', {
  strategy: 'jwt',
  accessToken: 'your token'
}, function(message, data) {
  console.log(message); // message will be null
  console.log(data); // data will be {"accessToken": "your token"}
  // You can now send authenticated messages to the server
});
```

OAuth1 Authentication



```
$ npm install feathers-authentication-oauth1 --save
```

[feathers-authentication-oauth1](#) is a server side module that allows you to use any [Passport](#) OAuth1 authentication strategy within your Feathers application, most notably [Twitter](#).

This module contains 2 core pieces:

1. The main initialization function
2. The `Verifier` class

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('feathers');
const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const oauth1 = require('feathers-authentication-oauth1');
const session = require('express-session');
const TwitterStrategy = require('passport-twitter').Strategy;
const app = feathers();

// Setup in memory session
app.use(session({
  secret: 'super secret',
  resave: true,
  saveUninitialized: true
}));

// Setup authentication
app.configure(authentication(settings));
app.configure(jwt());
app.configure(oauth1({
  name: 'twitter',
  Strategy: TwitterStrategy,
  consumerKey: '<your consumer key>',
  consumerSecret: '<your consumer secret>'
}));

// Setup a hook to only allow valid JWTs to authenticate
// and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Registering the OAuth1 plugin will automatically set up routes to handle the OAuth redirects and authorization.

Default Options

```
{
  idField: '<provider>Id', // The field to look up the entity by when logging in with the provider. Defaults to '<provider>Id' (ie. 'twitterId').
  path: '/auth/<provider>', // The route to register the middleware
  callbackURL: 'http(s)://hostame[:port]/auth/<provider>/callback', // The callback url. Will automatically take into account your host and port and whether you are in production based on your app environment to construct the url. (ie. in development http://localhost:3030/auth/twitter/callback)
  entity: 'user', // the entity that you are looking up
  service: 'users', // the service to look up the entity
  passReqToCallback: true, // whether the request object should be passed to `verify`
  session: true, // whether to use sessions,
  handler: function, // Express middleware for handling the oauth callback. Defaults to the built in middleware.
  formatter: function, // The response formatter. Defaults to the built in feathers-rest formatter, which returns JSON.
  Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for details.
}
```

Additional passport strategy options can be provided based on the OAuth1 strategy you are configuring.

Verifier

This is the verification class that handles the OAuth1 verification by looking up the entity (normally a `user`) on a given service and either creates or updates the entity and returns them. It has the following methods that can all be overridden. All methods return a promise except `verify`, which has the exact same signature as [passport-oauth1](#).

```
{
  constructor(app, options) // the class constructor
  _updateEntity(entity) // updates an existing entity
  _createEntity(entity) // creates an entity if they didn't exist already
  _normalizeResult(result) // normalizes result from service to account for pagination
  verify(req, accessToken, refreshToken, profile, done) // queries the service and calls the other internal functions.
}
```

Customizing the Verifier

The `Verifier` class can be extended so that you customize its behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want to use this plugin.

An example of customizing the Verifier:

```
import oauth1, { Verifier } from 'feathers-authentication-oauth1';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, accessToken, refreshToken, profile, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}
```

```
app.configure(oauth1({
  name: 'twitter'
  Strategy: TwitterStrategy,
  consumerKey: '<your consumer key>',
  consumerSecret: '<your consumer secret>',
  Verifier: CustomVerifier
}));
```

Customizing The OAuth Response

Whenever you authenticate with an OAuth1 provider such as Twitter, the provider sends back an `accessToken`, `refreshToken`, and a `profile` that contains the authenticated entity's information based on the OAuth1 `scopes` you have requested and been granted.

By default the `Verifier` takes everything returned by the provider and attaches it to the `entity` (ie. the user object) under the provider name. You will likely want to customize the data that is returned. This can be done by adding a `before` hook to both the `update` and `create` service methods on your `entity`'s service.

```
app.configure(oauth1({
  name: 'twitter',
  entity: 'user',
  service: 'users',
  Strategy,
  consumerKey: '<your consumer key>',
  consumerSecret: '<your consumer secret>'
}));

function customizeTwitterProfile() {
  return function(hook) {
    console.log('Customizing Twitter Profile');
    // If there is a twitter field they signed up or
    // signed in with twitter so let's pull the email. If
    if (hook.data.twitter) {
      hook.data.email = hook.data.twitter.email;
    }

    // If you want to do something whenever any OAuth
    // provider authentication occurs you can do this.
    if (hook.params.oauth) {
      // do something for all OAuth providers
    }

    if (hook.params.oauth.provider === 'twitter') {
      // do something specific to the twitter provider
    }

    return Promise.resolve(hook);
  };
}

app.service('users').hooks({
  before: {
    create: [customizeTwitterProfile()],
    update: [customizeTwitterProfile()]
  }
});
```

Client Usage

When this module is registered server side, whether you are using `feathers-authentication-client` or not you simply get the user to navigate to the authentication strategy url. This could be by setting `window.location` or through a link in your app.

For example you might have a login button for Twitter:

```
<a href="/auth/twitter" class="button">Login With Twitter</a>
```

OAuth2 Authentication



```
$ npm install feathers-authentication-oauth2 --save
```

[feathers-authentication-oauth2](#) is a server side module that allows you to use any [Passport](#) OAuth2 authentication strategy within your Feathers application. There are hundreds of them! Some commonly used ones are:

- [Facebook](#)
- [Instagram](#)
- [Github](#)
- [Google](#)
- [Spotify](#)

This module contains 2 core pieces:

1. The main initialization function
2. The `Verifier` class

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('feathers');
const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const oauth2 = require('feathers-authentication-oauth2');
const FacebookStrategy = require('passport-facebook').Strategy;
const app = feathers();

// Setup authentication
app.configure(authentication(settings));
app.configure(jwt());
app.configure(oauth2({
  name: 'facebook',
  Strategy: FacebookStrategy,
  clientID: '<your client id>',
  clientSecret: '<your client secret>',
  scope: ['public_profile', 'email']
}));

// Setup a hook to only allow valid JWTs to authenticate
// and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Registering the OAuth2 plugin will automatically set up routes to handle the OAuth redirects and authorization.

Default Options

```
{
  idField: '<provider>Id', // The field to look up the entity by when logging in with the provider. Defaults to '<provider>Id' (ie. 'facebookId').
  path: '/auth/<provider>', // The route to register the middleware
  callbackURL: 'http(s)://hostname[:port]/auth/<provider>/callback', // The callback url. Will automatically take into account your host and port and whether you are in production based on your app environment to construct the url. (ie. in development http://localhost:3030/auth/facebook/callback)
  successRedirect: undefined,
  failureRedirect: undefined,
  entity: 'user', // the entity that you are looking up
  service: 'users', // the service to look up the entity
  passReqToCallback: true, // whether the request object should be passed to `verify`
  session: false, // whether to use sessions,
  handler: function, // Express middleware for handling the oauth callback. Defaults to the built in middleware.
  formatter: function, // The response formatter. Defaults to the built in feathers-rest formatter, which returns JSON.
  Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for details.
}
```

Additional passport strategy options can be provided based on the OAuth1 strategy you are configuring.

Verifier

This is the verification class that handles the OAuth2 verification by looking up the entity (normally a `user`) on a given service and either creates or updates the entity and returns them. It has the following methods that can all be overridden. All methods return a promise except `verify`, which has the exact same signature as [passport-oauth2](#).

```
{
  constructor(app, options) // the class constructor
  _updateEntity(entity) // updates an existing entity
  _createEntity(entity) // creates an entity if they didn't exist already
  _normalizeResult(result) // normalizes result from service to account for pagination
  verify(req, accessToken, refreshToken, profile, done) // queries the service and calls the other internal functions.
}
```

Customizing the Verifier

The `Verifier` class can be extended so that you customize its behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want to use this plugin.

An example of customizing the Verifier:

```
import oauth2, { Verifier } from 'feathers-authentication-oauth2';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, accessToken, refreshToken, profile, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
}
```

```

    }
}

app.configure(oauth2({
  name: 'facebook'
  Strategy: FacebookStrategy,
  clientID: '<your client id>',
  clientSecret: '<your client secret>',
  scope: ['public_profile', 'email'],
  Verifier: CustomVerifier
}));
```

Customizing The OAuth Response

Whenever you authenticate with an OAuth2 provider such as Facebook, the provider sends back an `accessToken`, `refreshToken`, and a `profile` that contains the authenticated entity's information based on the OAuth2 `scopes` you have requested and been granted.

By default the `Verifier` takes everything returned by the provider and attaches it to the `entity` (ie. the `user` object) under the provider name. You will likely want to customize the data that is returned. This can be done by adding a `before` hook to both the `update` and `create` service methods on your `entity`'s service.

```

app.configure(oauth2({
  name: 'github',
  entity: 'user',
  service: 'users',
  Strategy,
  clientID: 'your client id',
  clientSecret: 'your client secret'
}));

function customizeGithubProfile() {
  return function(hook) {
    console.log('Customizing Github Profile');
    // If there is a github field they signed up or
    // signed in with github so let's pull the email. If
    if (hook.data.github) {
      hook.data.email = hook.data.github.email;
    }

    // If you want to do something whenever any OAuth
    // provider authentication occurs you can do this.
    if (hook.params.oauth) {
      // do something for all OAuth providers
    }

    if (hook.params.oauth.provider === 'github') {
      // do something specific to the github provider
    }

    return Promise.resolve(hook);
  };
}

app.service('users').hooks({
  before: {
    create: [customizeGithubProfile()],
    update: [customizeGithubProfile()]
  }
});
```

Client Usage

When this module is registered server side, whether you are using `feathers-authentication-client` or not you simply get the user to navigate to the authentication strategy url. This could be by setting `window.location` or through a link in your app.

For example you might have a login button for Facebook:

```
<a href="/auth/facebook" class="button">Login With Facebook</a>
```

Authentication hooks

[Star](#) 17 npm v0.1.4 [changelog .md](#)

```
$ npm install feathers-authentication-hooks --save
```

`feathers-authentication-hooks` is a package containing some useful hooks for authentication and authorization. For more information about hooks, refer to the [chapter on hooks](#).

Note: Restricting authentication hooks will only run when `params.provider` is set (as in when the method is accessed externally through a transport like [REST](#) or [Socketio](#)).

queryWithCurrentUser

The `queryWithCurrentUser` **before** hook will automatically add the user's `id` as a parameter in the query. This is useful when you want to only return data, for example "messages", that were sent by the current user.

```
const hooks = require('feathers-authentication-hooks');

app.service('messages').before({
  find: [
    hooks.queryWithCurrentUser({ idField: 'id', as: 'sentBy' })
  ]
});
```

Options

- `idField` (default: `'_id'`) [optional] - The id field on your user object.
- `as` (default: `'userId'`) [optional] - The id field for a user on the resource you are requesting.

When using this hook with the default options the `user._id` will be copied into `hook.params.query.userId`.

restrictToOwner

`restrictToOwner` is meant to be used as a **before** hook. It only allows the user to retrieve or modify resources that are owned by them. It will return a *Forbidden* error without the proper permissions. It can be used on *any* method.

For `find` method calls and `patch`, `update` and `remove` of many (with `id` set to `null`), the `queryWithCurrentUser` hook will be called to limit the query to the current user. For all other cases it will retrieve the record and verify the owner before continuing.

```
const hooks = require('feathers-authentication-hooks');

app.service('messages').before({
  remove: [
    hooks.restrictToOwner({ idField: 'id', ownerField: 'sentBy' })
  ]
});
```

Options

- `idField` (default: '_id') [optional] - The id field on your user object.
- `ownerField` (default: 'userId') [optional] - The id field for a user on your resource.

restrictToAuthenticated

The `restrictToAuthenticated` hook throws an error if there isn't a logged-in user by checking for the `hook.params.user` object. It can be used on **any** service method and is intended to be used as a **before** hook. It doesn't take any arguments.

```
const hooks = require('feathers-authentication-hooks');

app.service('user').before({
  get: [
    hooks.restrictToAuthenticated()
  ]
});
```

Options

- `entity` (default: 'user') [optional] - The property name on `hook.params` to check for

associateCurrentUser

The `associateCurrentUser` **before** hook is similar to the `queryWithCurrentUser`, but works on the incoming `data` instead of the `query` params. It's useful for automatically adding the userId to any resource being created. It can be used on `create`, `update`, or `patch` methods.

```
const hooks = require('feathers-authentication-hooks');

app.service('messages').before({
  create: [
    hooks.associateCurrentUser({ idField: 'id', as: 'sentBy' })
  ]
});
```

Options

- `idField` (default: '_id') [optional] - The id field on your user object.
- `as` (default: 'userId') [optional] - The id field for a user that you want to set on your resource.

restrictToRoles

`restrictToRoles` is meant to be used as a **before** hook. It only allows the user to retrieve resources that are owned by them or protected by certain roles. It will return a *Forbidden* error without the proper permissions. It can be used on `all` methods when the `owner` option is set to 'false'. When the `owner` option is set to `true` the hook can only be used on `get`, `update`, `patch`, and `remove` service methods.

```
const hooks = require('feathers-authentication-hooks');

app.service('messages').before({
  remove: [
    hooks.restrictToRoles({
      roles: ['admin', 'super-admin'],
    })
  ]
});
```

```

        fieldName: 'permissions',
        idField: 'id',
        ownerField: 'sentBy',
        owner: true
    })
]
});

```

Options

- `roles` (**required**) - An array of roles that a user must have at least one of in order to access the resource.
- `fieldName` (default: 'roles') [optional] - The field on your user object that denotes their roles.
- `idField` (default: '_id') [optional] - The id field on your user object.
- `ownerField` (default: 'userId') [optional] - The id field for a user on your resource.
- `owner` (default: 'false') [optional] - Denotes whether it should also allow owners regardless of their role (ie. the user has the role **or** is an owner).

hasRoleOrRestrict

`hasRoleOrRestrict` is meant to be used as a **before** hook for any service on the **find** or **get** methods. Unless the user has one of the roles provided, it will add a restriction onto the query to limit what resources return.

```

const hooks = require('feathers-authentication-hooks');

app.service('messages').before({
  find: [
    hooks.hasRoleOrRestrict({
      roles: ['admin', 'super-admin'],
      fieldName: 'permissions',
      restrict: { approved: true }
    })
  ]
});

```

Options

- `roles` (**required**) - An array of roles that a user must have at least one of in order to access the resource.
- `fieldName` (default: 'roles') [optional] - The field on your user object that denotes their roles.
- `restrict` (default: undefined) - The query to merge into the client query to limit what resources are accessed

Common API

All database adapters implement a common interface for initialization, pagination, extending and querying. This chapter describes the common adapter initialization and options, how to enable and use pagination, the details on how specific service methods behave and how to extend an adapter with custom functionality.

Important: Every database adapter is an implementation of the [Feathers service interface](#). We recommend being familiar with services, service events and hooks before using a database adapter.

Initialization

service([options])

Returns a new service instance initialized with the given options.

```
const service = require('feathers-<adaptername>');

app.use('/messages', service());
app.use('/messages', service({ id, events, paginate }));
```

Options:

- `id` (*optional*) - The name of the id field property (usually set by default to `id` or `_id`).
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

Pagination

When initializing an adapter you can set the following pagination options in the `paginate` object:

- `default` - Sets the default number of items when `$limit` is not set
- `max` - Sets the maximum allowed number of items per page (even if the `$limit` query parameter is set higher)

When `paginate.default` is set, `find` will return an *page object* (instead of the normal array) in the following form:

```
{
  "total": "<total number of records>",
  "limit": "<max number of items per page>",
  "skip": "<number of skipped items (offset)>",
  "data": [/* data */]
}
```

The pagination options can be set as follows:

```
const service = require('feathers-<db-name>');

// Set the `paginate` option during initialization
app.use('/todos', service({
  paginate: {
    default: 5,
    max: 25
  }
}));
```

```
// override pagination in `params.paginate` for this call
app.service('todos').find({
  paginate: {
    default: 100,
    max: 200
  }
});

// disable pagination for this call
app.service('todos').find({
  paginate: false
});
```

Note: Disabling or changing the default pagination is not available in the client. Only `params.query` is passed to the server (also see a [workaround here](#))

Pro tip: To just get the number of available records set `$limit` to `0`. This will only run a (fast) counting query against the database.

Service methods

This section describes specifics on how the [service methods](#) are implemented for all adapters.

adapter.find(params) -> Promise

Returns a list of all records matching the query in `params.query` using the [common querying mechanism](#). Will either return an array with the results or a page object if [pagination is enabled](#).

Important: When used via REST URLs all query values are strings. Depending on the database the values in `params.query` might have to be converted to the right type in a [before hook](#).

```
// Find all messages for user with id 1
app.service('messages').find({
  query: {
    userId: 1
  }
}).then(messages => console.log(messages));

// Find all messages belonging to room 1 or 3
app.service('messages').find({
  query: {
    roomId: {
      $in: [ 1, 3 ]
    }
  }
}).then(messages => console.log(messages));
```

Find all messages for user with id 1

```
GET /messages?userId=1
```

Find all messages belonging to room 1 or 3

```
GET /messages?roomId[$in]=1&roomId[$in]=3
```

adapter.get(id, params) -> Promise

Retrieve a single record by its unique identifier (the field set in the `id` option during initialization).

```
app.service('messages').get(1)
  .then(message => console.log(message));
```

GET /messages/1

adapter.create(data, params) -> Promise

Create a new record with `data`. `data` can also be an array to create multiple records.

```
app.service('messages').create({
  text: 'A test message'
})
  .then(message => console.log(message));

app.service('messages').create([
  {
    text: 'Hi'
  },
  {
    text: 'How are you'
  }
])
  .then(messages => console.log(messages));
```

```
POST /messages
{
  "text": "A test message"
}
```

adapter.update(id, data, params) -> Promise

Completely replaces a single record identified by `id` with `data`. Does not allow replacing multiple records (`id` can't be `null`). `id` can not be changed.

```
app.service('messages').update(1, {
  text: 'Updates message'
})
  .then(message => console.log(message));
```

```
PUT /messages/1
{ "text": "Updated message" }
```

adapter.patch(id, data, params) -> Promise

Merges a record identified by `id` with `data`. `id` can be `null` to allow replacing multiple records (all records that match `params.query` the same as in `.find`). `id` can not be changed.

```
app.service('messages').update(1, {
  text: 'A patched message'
})
  .then(message => console.log(message));

const params = {
  query: { read: false }
};

// Mark all unread messages as read
app.service('messages').patch(null, {
  read: true
}, params);
```

```
PATCH /messages/1
{ "text": "A patched message" }
```

Mark all unread messages as read

```
PATCH /messages?read=false
{ "read": true }
```

adapter.remove(id, params) -> Promise

Removes a record identified by `id`. `id` can be `null` to allow removing multiple records (all records that match `params.query` the same as in `.find`).

```
app.service('messages').remove(1)
  .then(message => console.log(message));

const params = {
  query: { read: true }
};

// Remove all read messages
app.service('messages').remove(null, params);
```

```
DELETE /messages/1
```

Remove all read messages

```
DELETE /messages?read=true
```

Extending Adapters

There are two ways to extend existing database adapters. Either by extending the ES6 base class or by adding functionality through hooks.

ProTip: Keep in mind that calling the original service methods will return a Promise that resolves with the value.

Hooks

The most flexible option is weaving in functionality through [hooks](#). For example, `createdAt` and `updatedAt` timestamps could be added like this:

```
const feathers = require('feathers');
const hooks = require('feathers-hooks');

// Import the database adapter of choice
const service = require('feathers-<adapter>');

const app = feathers()
  .configure(hooks())
  .use('/todos', service({
    paginate: {
      default: 2,
      max: 4
    }
  })
}
```

```
});  
  
app.service('todos').hooks({  
  before: {  
    create: [  
      (hook) => hook.data.createdAt = new Date()  
    ],  
  
    update: [  
      (hook) => hook.data.updatedAt = new Date()  
    ]  
  }  
});  
  
app.listen(3030);
```

Classes (ES6)

All modules also export an [ES6 class](#) as `Service` that can be directly extended like this:

```
'use strict';  
  
const Service = require('feathers-<database>').Service;  
  
class MyService extends Service {  
  create(data, params) {  
    data.created_at = new Date();  
  
    return super.create(data, params);  
  }  
  
  update(id, data, params) {  
    data.updated_at = new Date();  
  
    return super.update(id, data, params);  
  }  
}  
  
app.use('/todos', new MyService({  
  paginate: {  
    default: 2,  
    max: 4  
  }  
}));
```

Querying

All official database adapters support a common way for querying, sorting, limiting and selecting `find` method calls as part of `params.query`. Querying also applies `update`, `patch` and `remove` method calls if the `id` is set to `null`.

Important: When used via REST URLs all query values are strings. Depending on the service the values in `params.query` might have to be converted to the right type in a [before hook](#).

Equality

All fields that do not contain special query parameters are compared directly for equality.

```
// Find all unread messages in room #2
app.service('messages').find({
  query: {
    read: false,
    roomId: 2
  }
});
```

GET /messages?read=false&roomId=2

\$limit

`$limit` will return only the number of results you specify:

```
// Retrieves the first two unread messages
app.service('messages').find({
  query: {
    $limit: 2,
    read: false
  }
});
```

GET /messages?\$limit=2&read=false

Pro tip: With [pagination enabled](#), to just get the number of available records set `$limit` to `0`. This will only run a (fast) counting query against the database and return a page object with the `total` and an empty `data` array.

\$skip

`$skip` will skip the specified number of results:

```
// Retrieves the next two unread messages
app.service('messages').find({
  query: {
    $limit: 2,
    $skip: 2,
    read: false
  }
});
```

```
});
```

```
GET /messages?$limit=2&$skip=2&read=false
```

\$sort

`$sort` will sort based on the object you provide. It can contain a list of properties by which to sort mapped to the order (`1` ascending, `-1` descending).

```
// Find the 10 newest messages
app.service('messages').find({
  query: {
    $limit: 10,
    $sort: {
      createdAt: -1
    }
  }
});
```

```
/messages?$limit=10&$sort[createdAt]=-1
```

\$select

`$select` allows to pick which fields to include in the result. This will work for any service method.

```
// Only return the `text` and `userId` field in a message
app.service('messages').find({
  query: {
    $select: [ 'text', 'userId' ]
  }
});

app.service('messages').get(1, {
  query: {
    $select: [ 'text' ]
  }
});
```

```
GET /messages?$select=text&$select=userId
GET /messages/1?$select=text
```

To exclude fields from a result the [remove hook](#) can be used.

\$in , \$nin

Find all records where the property does (`$in`) or does not (`$nin`) match any of the given values.

```
// Find all messages in room 2 or 5
app.service('messages').find({
  query: {
    roomId: {
      $in: [ 2, 5 ]
    }
  }
});
```

```
});
```

```
GET /messages?roomId[$in]=2&roomId[$in]=5
```

\$lt , \$lte

Find all records where the value is less (`$lt`) or less and equal (`$lte`) to a given value.

```
// Find all messages older than a day
const DAY_MS = 24 * 60 * 60 * 1000;

app.service('messages').find({
  query: {
    createdAt: {
      $lt: new Date().getTime() - DAY_MS
    }
  }
});
```

```
GET /messages?createdAt[$lt]=1479664146607
```

\$gt , \$gte

Find all records where the value is more (`$gt`) or more and equal (`$gte`) to a given value.

```
// Find all messages within the last day
const DAY_MS = 24 * 60 * 60 * 1000;

app.service('messages').find({
  query: {
    createdAt: {
      $gt: new Date().getTime() - DAY_MS
    }
  }
});
```

```
GET /messages?createdAt[$gt]=1479664146607
```

\$ne

Find all records that do not equal the given property value.

```
// Find all messages that are not marked as archived
app.service('messages').find({
  query: {
    archived: {
      $ne: true
    }
  }
});
```

```
GET /messages?archived[$ne]=true
```

\$or

Find all records that match any of the given criteria.

```
// Find all messages that are not marked as archived
// or any message from room 2
app.service('messages').find({
  query: {
    $or: [
      { archived: { $ne: true } },
      { roomId: 2 }
    ]
  }
});
```

```
GET /messages?$or[0][archived][$ne]=true&$or[1][roomId]=2
```

\$search

`$search` is not a public API, but may be added through hooks. For example, hooks exists for NeDB and MongoDB:

- [feathers-nedb-fuzzy-search](#)
- [feathers-mongodb-fuzzy-search](#)
- [feathers-solr](#)

Example usage:

```
// Find all messages that contains the text 'hello'
app.service('messages').find({
  query: {
    $search: 'hello'
  }
});
```

```
GET /messages?$search=hello
```

In Memory

[Star](#) 17 npm v1.2.1 changelog .md

[feathers-memory](#) is a database service adapter for in-memory data storage that works on all platforms.

```
$ npm install --save feathers-memory
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

API

service([options])

Returns a new service instance initialized with the given options.

```
const service = require('feathers-memory');

app.use('/messages', service());
app.use('/messages', service({ id, startId, store, events, paginate }));
```

Options:

- `id` (*optional*, default: `'id'`) - The name of the id field property.
- `startId` (*optional*, default: `0`) - An id number to start with that will be incremented for every new record (unless it is already set).
- `store` (*optional*) - An object with id to item assignments to pre-initialize the data store
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

Example

Here is an example of a Feathers server with a `messages` in-memory service that supports pagination:

```
$ npm install feathers body-parser feathers-rest feathers-socketio feathers-memory feathers-errors
```

In `app.js`:

```
const feathers = require('feathers');
const bodyParser = require('body-parser');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const memory = require('feathers-memory');
const errorHandler = require('feathers-errors/handler');

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable REST services
  .configure(socketio())
```

```
// Turn on JSON parser for REST services
.use(bodyParser.json())
// Turn on URL-encoded parser for REST services
.use(bodyParser.urlencoded({ extended: true }))
// Create an in-memory Feathers service with a default page size of 2 items
// and a maximum size of 4
.use('/messages', memory({
  paginate: {
    default: 2,
    max: 4
  }
}))
// Set up default error handler
.use(errorHandler());

// Create a dummy Message
app.service('messages').create({
  text: 'Message created on server'
}).then(message => console.log('Created message', message));

// Start the server.
const port = 3030;

app.listen(port, () => {
  console.log(`Feathers server listening on port ${port}`)
});
```

Run the example with `node app` and go to localhost:3030/messages.

NeDB



[feathers-nedb](#) is a database service adapter for [NeDB](#), an embedded datastore with a [MongoDB](#) like API. NeDB can store data in-memory or on the filesystem which makes it useful as a persistent storage without a separate database server.

```
$ npm install --save nedb feathers-nedb
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

API

service(options)

Returns a new service instance initialized with the given options. `Model` has to be an NeDB database instance.

```
const NeDB = require('nedb');
const service = require('feathers-nedb');

// Create a NeDB instance
const Model = new NeDB({
  filename: './data/messages.db',
  autoload: true
});

app.use('/messages', service({ Model }));
app.use('/messages', service({ Model, id, events, paginate }));
```

Options:

- `Model` (**required**) - The NeDB database instance. See the [NeDB API](#) for more information.
- `id` (*optional*, default: `'_id'`) - The name of the id field property. By design, NeDB will always add an `_id` property.
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

params.nedb

When making a [service method](#) call, `params` can contain an `nedb` property which allows to pass additional [NeDB options](#), for example to allow `upsert`:

```
app.service('messages').update('someid', {
  text: 'This message will be either created or updated'
}, {
  nedb: { upsert: true }
});
```

Example

Here is an example of a Feathers server with a `messages` NeDB service that supports pagination and persists to `db-data/messages`:

```
$ npm install feathers feathers-errors feathers-rest feathers-socketio feathers-nedb nedb body-parser
```

In `app.js`:

```
const NeDB = require('nedb');
const feathers = require('feathers');
const errorHandler = require('feathers-errors/handler');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const service = require('feathers-nedb');

const db = new NeDB({
  filename: './db-data/messages',
  autoload: true
});

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}))
  // Connect to the db, create and register a Feathers service.
  .use('/messages', service({
    Model: db,
    paginate: {
      default: 2,
      max: 4
    }
  }))
  // Set up default error handler
  .use(errorHandler());

// Create a dummy Message
app.service('messages').create({
  text: 'Message created on server'
}).then(message => console.log('Created message', message));

// Start the server.
const port = 3030;

app.listen(port, () => {
  console.log(`Feathers server listening on port ${port}`);
});
```

Run the example with `node app` and go to localhost:3030/messages.

LocalStorage and AsyncStorage

[Star](#) 14 npm v1.0.1 changelog .md

[feathers-localstorage](#) is a database service adapter that extends [feathers-memory](#) and stores data in [localStorage](#) in the browser or [AsyncStorage](#) in React Native.

```
$ npm install --save feathers-localstorage
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

API

service(options)

Returns a new service instance initialized with the given options.

```
const service = require('feathers-localstorage');

app.use('/messages', service({
  storage: window.localStorage || AsyncStorage
}));
app.use('/messages', service({ storage, id, startId, name, store, paginate }));
```

Options:

- `storage` (**required**) - The local storage engine. You can pass in the browsers `window.localStorage`, React Native's `AsyncStorage` or a NodeJS `localStorage` module.
- `id` (*optional*, default: `'id'`) - The name of the id field property.
- `startId` (*optional*, default: `0`) - An id number to start with that will be incremented for new record.
- `name` (*optional*, default: `'feathers'`) - The key to store data under in local or async storage.
- `store` (*optional*) - An object with id to item assignments to pre-initialize the data store
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

Example

See the [clients](#) chapter for more information about using Feathers in the browser and React Native.

Browser

```
<script type="text/javascript" src="socket.io/socket.io.js"></script>
<script type="text/javascript" src="//unpkg.com/feathers-client@^2.0.0/dist/feathers.js"></script>
<script type="text/javascript" src="//unpkg.com/feathers-localstorage@^1.0.0/dist/localstorage.js"></script>
<script type="text/javascript">
  var service = feathers.localstorage({
    storage: window.localStorage
  });
  var app = feathers().use('/messages', service);

  var messages = app.service('messages');
```

```
messages.on('created', function(message) {
  console.log('Someone created a message', message);
});

messages.create({
  text: 'Message created in browser'
});
</script>
```

React Native

```
$ npm install feathers feathers-localstorage --save
```

```
import React from 'react-native';
import localstorage from 'feathers-localstorage';
import feathers from 'feathers';

const { AsyncStorage } = React;

const app = feathers()
  .use('/messages', localstorage({ storage: AsyncStorage }));

const messages = app.service('messages');

messages.on('created', function(message) {
  console.log('Someone created a message', message);
});

messages.create({
  text: 'Message from React Native'
});
```

MongoDB

[Star](#) 35 npm v2.9.0 [changelog .md](#)

[feathers-mongodb](#) is a database adapter for [MongoDB](#). It uses the [official NodeJS driver for MongoDB](#).

```
$ npm install --save mongodb feathers-mongodb
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

This adapter also requires a [running MongoDB](#) database server.

API

service(options)

Returns a new service instance initialized with the given options. `Model` has to be a MongoDB collection.

```
const MongoClient = require('mongodb').MongoClient;
const service = require('feathers-mongodb');

MongoClient.connect('mongodb://localhost:27017/feathers').then(db => {
  app.use('/messages', service({
    Model: db.collection('messages')
  }));
  app.use('/messages', service({ Model, id, events, paginate }));
});
```

Options:

- `Model` (**required**) - The MongoDB collection instance
- `id` (*optional*, default: `'_id'`) - The name of the id field property. By design, MongoDB will always add an `_id` property.
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

params.mongodb

When making a [service method](#) call, `params` can contain an `mongodb` property (for example, `{upsert: true}`) which allows to modify the options used to run the MongoDB query.

Example

Here is an example of a Feathers server with a `messages` endpoint that writes to the `feathers` database and the `messages` collection.

```
$ npm install feathers feathers-errors feathers-rest feathers-socketio feathers-mongodb mongodb body-parser
```

In `app.js`:

```

const feathers = require('feathers');
const errorHandler = require('feathers-errors/handler');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const MongoClient = require('mongodb').MongoClient;
const service = require('feathers-mongodb');

// Create a feathers instance.
const app = feathers()
  // Enable Socket.io
  .configure(socketio())
  // Enable REST services
  .configure(rest())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}));

// Connect to your MongoDB instance(s)
MongoClient.connect('mongodb://localhost:27017/feathers').then(function(db){
  // Connect to the db, create and register a Feathers service.
  app.use('/messages', service({
    Model: db.collection('messages'),
    paginate: {
      default: 2,
      max: 4
    }
  }));
  // A basic error handler, just like Express
  app.use(errorHandler());
  // Create a dummy Message
  app.service('messages').create({
    text: 'Message created on server'
  }).then(message => console.log('Created message', message));
  // Start the server.
  const port = 3030;
  app.listen(port, () => {
    console.log(`Feathers server listening on port ${port}`);
  });
}).catch(error => console.error(error));

```

Run the example with `node app` and go to localhost:3030/messages.

Querying

Additionally to the [common querying mechanism](#) this adapter also supports [MongoDB's query syntax](#) and the `update` method also supports MongoDB [update operators](#).

Important: External query values (especially through URLs) may have to be converted to the same type stored in MongoDB in a before `hook` otherwise no matches will be found.

For example, a `find` call for `_id` (which is a MongoDB object id) and `age` (which is a number) a hook like this can be used:

```

const ObjectId = require('mongodb').ObjectId;

app.service('users').hooks({
  before: {
    find(hook) {

```

```

const { query = {} } = hook.params;

if(query._id) {
  query._id = new ObjectId(query._id);
}

if(query.age !== undefined) {
  query.age = parseInt(query.age, 10);
}

hook.params.query = query;

return Promise.resolve(hook);
}
);

```

Which will allow queries like `/users?_id=507f1f77bcf86cd799439011&age=25`.

Collation Support

This adapter includes support for [collation and case insensitive indexes available in MongoDB v3.4](#). Collation parameters may be passed using the special `collation` parameter to the `find()`, `remove()` and `patch()` methods.

Example: Patch records with case-insensitive alphabetical ordering.

The example below would patch all student records with grades of `'c'` or `'C'` and above (a natural language ordering). Without collations this would not be as simple, since the comparison `{ $gt: 'c' }` would not include uppercase grades of `'C'` because the code point of `'c'` is less than that of `'C'`.

```

const patch = { shouldStudyMore: true };
const query = { grade: { $gte: 'c' } };
const collation = { locale: 'en', strength: 1 };
students.patch(null, patch, { query, collation }).then( ... );

```

Example: Find records with a case-insensitive search.

Similar to the above example, this would find students with a grade of `'c'` or greater, in a case-insensitive manner.

```

const query = { grade: { $gte: 'c' } };
const collation = { locale: 'en', strength: 1 };
students.find({ query, collation }).then( ... );

```

For more information on MongoDB's collation feature, visit the [collation reference page](#).

Mongoose



[feathers-mongoose](#) is a database adapter for [Mongoose](#), an object modeling tool for [MongoDB](#).

```
$ npm install --save mongoose feathers-mongoose
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

This adapter also requires a [running MongoDB](#) database server.

API

service(options)

Returns a new service instance initialized with the given options. `Model` has to be a Mongoose model. See the [Mongoose Guide](#) for more information on defining your model.

```
const mongoose = require('mongoose');
const service = require('feathers-mongoose');

// A module that exports your Mongoose model
const Model = require('./models/message');

// Make Mongoose use the ES6 promise
mongoose.Promise = global.Promise;

// Connect to a local database called `feathers`
mongoose.connect('mongodb://localhost:27017/feathers');

app.use('/messages', service({ Model }));
app.use('/messages', service({ Model, lean, id, events, paginate }));
```

Options:

- `Model` (**required**) - The Mongoose model definition
- `lean` (*optional*, default: `true`) - Runs queries faster by returning plain objects instead of Mongoose models.
- `id` (*optional*, default: `'_id'`) - The name of the id field property.
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size
- `discriminators` (*optional*) - A list of mongoose models that inherit from `Model`.

Important: To avoid odd error handling behaviour, always set `mongoose.Promise = global.Promise`. If not available already, Feathers comes with a polyfill for native Promises.

Important: When setting `lean` to `false`, Mongoose models will be returned which can not be modified unless they are converted to a regular JavaScript object via `toObject`.

Note: You can get access to the Mongoose model via `this.Model` inside a [hook](#) and use it as usual. See the [Mongoose Guide](#) for more information on defining your model.

params.mongoose

When making a `service method` call, `params` can contain a `mongoose` property which allows you to modify the options used to run the Mongoose query. Normally, this will be set in a before `hook`:

```
app.service('messages').hooks({
  before: [
    patch(hook) {
      // Set some additional Mongoose options
      // The adapter tries to use sane defaults
      // but they can always be changed here
      hook.params.mongoose = {
        runValidators: true,
        setDefaultsOnInsert: true
      }
    }
  ]
});
```

The `mongoose` property is also useful for performing upserts on a `patch` request. "Upserts" do an update if a matching record is found, or insert a record if there's no existing match. The following example will create a document that matches the `data`, or if there's already a record that matches the `params.query`, that record will be updated.

```
const data = { address: '123', identifier: 'my-identifier' }
const params = {
  query: { address: '123' },
  mongoose: { upsert: true }
}
app.service('address-meta').patch(null, data, params)
```

Example

Here's a complete example of a Feathers server with a `messages` Mongoose service.

```
$ npm install feathers feathers-errors feathers-rest body-parser mongoose feathers-mongoose
```

In `message-model.js`:

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;
const MessageSchema = new Schema({
  text: {
    type: String,
    required: true
  }
});
const Model = mongoose.model('Message', MessageSchema);

module.exports = Model;
```

Then in `app.js`:

```
const feathers = require('feathers');
const errorHandler = require('feathers-errors/handler');
const rest = require('feathers-rest');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const service = require('feathers-mongoose');

const Model = require('./message-model');
```

```
// Tell mongoose to use native promises
// See http://mongoosejs.com/docs/promises.html
mongoose.Promise = global.Promise;

// Connect to your MongoDB instance(s)
mongoose.connect('mongodb://localhost:27017/feathers');

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}))
  // Connect to the db, create and register a Feathers service.
  .use('/messages', service({
    Model,
    lean: true, // set to false if you want Mongoose documents returned
    paginate: {
      default: 2,
      max: 4
    }
  }));
  .use(errorHandler());

// Create a dummy Message
app.service('messages').create({
  text: 'Message created on server'
}).then(function(message) {
  console.log('Created message', message);
});

// Start the server.
const port = 3030;
app.listen(port, () => {
  console.log(`Feathers server listening on port ${port}`);
});
```

You can run this example by using `node app` and go to localhost:3030/messages.

Querying, Validation

Mongoose by default gives you the ability to add [validations at the model level](#). Using an error handler like the one that [comes with Feathers](#) your validation errors will be formatted nicely right out of the box!

For more information on querying and validation refer to the [Mongoose documentation](#).

\$populate

For Mongoose, the special `$populate` query parameter can be used to allow [Mongoose query population](#).

```
app.service('posts').find({
  query: { $populate: 'user' }
});
```

Discriminators (Inheritance)

Instead of strict inheritance, Mongoose uses [discriminators](#) as their schema inheritance model. To use them, pass in a `discriminatorKey` option to your schema object and use `Model.discriminator('modelName', schema)` instead of `mongoose.model()`

Feathers comes with full support for mongoose discriminators, allowing for automatic fetching of inherited types. A typical use case might look like:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var Post = require('../post');
var feathers = require('feathers');
var app = feathers();
var service = require('feathers-mongoose');

// Discriminator key, we'll use this later to refer to all text posts
var options = {
  discriminatorKey: '_type'
};

var TextPostSchema = new Schema({
  text: { type: String, default: null }
}, options);

TextPostSchema.index({'updatedAt': -1, background: true});

// Note the use of `Post.discriminator` rather than `mongoose.discriminator`.
var TextPost = Post.discriminator('text', TextPostSchema);

// Using the discriminators option, let feathers know about any inherited models you may have
// for that service
app.use('/posts', service({
  Model: Post,
  discriminators: [TextPost]
}))
```

Without support for discriminators, when you perform a `.get` on the posts service, you'd only get back `Post` models, not `TextPost` models. Now in your query, you can specify a value for your `discriminatorKey`:

```
{
  _type: 'text'
}
```

and Feathers will automatically swap in the correct model and execute the query it instead of its parent model.

Sequelize



[feathers-sequelize](#) is a database adapter for [Sequelize](#), an ORM for Node.js. It supports PostgreSQL, MySQL, MariaDB, SQLite and MSSQL and features transaction support, relations, read replication and more.

```
npm install --save feathers-sequelize
```

And [one of the following](#):

```
npm install --save pg pg-hstore
npm install --save mysql // For both mysql and mariadb dialects
npm install --save sqlite3
npm install --save tedious // MSSQL
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

For more information about models and general Sequelize usage, follow up in the [Sequelize documentation](#).

A quick note about raw queries

By default, all `feathers-sequelize` operations will return `raw` data (using `raw: true` when querying the database). This results in faster execution and allows `feathers-sequelize` to interoperate with `feathers-common` hooks and other 3rd party integrations. However, this will bypass some of the "goodness" you get when using Sequelize as an ORM:

- custom getters/setters will be bypassed
- model-level validations are bypassed
- associated data loads a bit differently
- ...and several other issues that one might not expect

Don't worry! The solution is easy. Please read the guides about [working with model instances](#).

API

service(options)

Returns a new service instance initialized with the given options.

```
const Model = require('./models/mymodel');
const service = require('feathers-sequelize');

app.use('/messages', service({ Model }));
app.use('/messages', service({ Model, id, events, paginate }));
```

Options:

- `Model` (**required**) - The Sequelize model definition
- `id` (*optional*, default: `'_id'`) - The name of the id field property.
- `raw` (*optional*, default: `true`) - Runs queries faster by returning plain objects instead of Sequelize models.

- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

params.sequelize

When making a [service method](#) call, `params` can contain an `sequelize` property which allows to pass additional Sequelize options. This can e.g. be used to [retrieve associations](#). Normally this will be set in a before [hook](#):

```
app.service('messages').hooks({
  before: [
    find(hook) {
      // Get the Sequelize instance. In the generated application via:
      const sequelize = hook.app.get('sequelizeClient');

      hook.paramssequelize = {
        include: [ User ]
      }
    }
  ]
});
```

Example

Here is an example of a Feathers server with a `messages` SQLite Sequelize Model:

```
$ npm install feathers feathers-errors feathers-rest feathers-socketio body-parser sequelize feathers-sequelize
sqlite3
```

In `app.js`:

```
const path = require('path');
const feathers = require('feathers');
const errorHandler = require('feathers-errors/handler');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const Sequelize = require('sequelize');
const service = require('feathers-sequelize');

const sequelize = new Sequelize('sequelize', '', '', {
  dialect: 'sqlite',
  storage: path.join(__dirname, 'db.sqlite'),
  logging: false
});
const Message = sequelize.define('message', {
  text: {
    type: Sequelize.STRING,
    allowNull: false
  }
}, {
  freezeTableName: true
});

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
```

```
// Turn on URL-encoded parser for REST services
.use(bodyParser.urlencoded({ extended: true }))
// Create an in-memory Feathers service with a default page size of 2 items
// and a maximum size of 4
.use('/messages', service({
  Model: Message,
  paginate: {
    default: 2,
    max: 4
  }
}))
.use(errorHandler());

Message.sync({ force: true }).then(() => {
  // Create a dummy Message
  app.service('messages').create({
    text: 'Message created on server'
  }).then(message => console.log('Created message', message.toJSON()));
});

// Start the server
const port = 3030;

app.listen(port, () => {
  console.log(`Feathers server listening on port ${port}`);
});
```

Run the example with `node app` and go to localhost:3030/messages.

Querying

Additionally to the [common querying mechanism](#) this adapter also supports all [Sequelize query operators](#).

Associations and relations

Follow up in the [Sequelize documentation for associations](#), [this issue](#) and [this Stackoverflow answer](#).

Working with Sequelize Model instances

It is highly recommended to use `raw` queries, which is the default. However, there are times when you will want to take advantage of [Sequelize Instance](#) methods. There are two ways to tell feathers to return Sequelize instances:

1. Set `{ raw: false }` in a "before" hook:

```
function rawFalse(hook) {
  if (!hook.params.sequelize) hook.params.sequelize = {};
  Object.assign(hook.params.sequelize, { raw: false });
  return hook;
}
hooks.before.find = [rawFalse];
```

2. Use the new `hydrate` hook in the "after" phase:

```
const hydrate = require('feathers-sequelize/hooks/hydrate');
hooks.after.find = [hydrate()];

// Or, if you need to include associated models, you can do the following:
function includeAssociated (hook) {
  return hydrate({
```

```

        include: [{ model: hook.app.services fooservice.Model }]
    }).call(this, hook);
}
hooks.after.find = [includeAssociated];

```

For a more complete example see this [gist](#).

Important: When working with Sequelize Instances, most of the feathers-hooks-common will no longer work. If you need to use a common hook or other 3rd party hooks, you should use the "dehydrate" hook to convert data back to a plain object:

```

const hydrate = require('feathers-sequelize/hooks/hydrate');
const dehydrate = require('feathers-sequelize/hooks/dehydrate');
const { populate } = require('feathers-hooks-common');

hooks.after.find = [hydrate(), doSomethingCustom(), dehydrate(), populate()];

```

Validation

Sequelize by default gives you the ability to [add validations at the model level](#). Using an error handler like the one that [comes with Feathers](#) your validation errors will be formatted nicely right out of the box!

Migrations

Migrations with feathers and sequelize are quite simple. This guide will walk you through creating the recommended file structure, but you are free to rearrange things as you see fit. The following assumes you have a `migrations` folder in the root of your app.

Initial Setup: one-time tasks

- Install the [sequelize CLI](#):

```
npm install sequelize-cli --save -g
```

- Create a `.sequelizerc` file in your project root with the following content:

```

const path = require('path');

module.exports = {
  'config': path.resolve('migrations/config/config.js'),
  'migrations-path': path.resolve('migrations'),
  'seeders-path': path.resolve('migrations/seeders'),
  'models-path': path.resolve('migrations/models')
};

```

- Create the migrations config in `migrations/config/config.js` :

```

const app = require('../src/app');
const env = process.env.NODE_ENV || 'development';
const dialect = 'mysql'|'sqlite'|'postgres'|'mssql';

module.exports = [
  [env]: {
    dialect,
    url: app.get(dialect),

```

```

    migrationStorageTableName: '_migrations'
}
};

```

- Define your models config in `migrations/models/index.js` :

```

const Sequelize = require('sequelize');
const app = require('../src/app');
const sequelize = app.get('sequelizeClient');
const models = sequelize.models;

// The export object must be a dictionary of model names -> models
// It must also include sequelize (instance) and Sequelize (constructor) properties
module.exports = Object.assign({
  Sequelize,
  sequelize
}, models);

```

Migrations workflow

The migration commands will load your application and it is therefore required that you define the same environment variables as when running your application. For example, many applications will define the database connection string in the startup command:

```
DATABASE_URL=postgres://user:pass@host:port/dbname npm start
```

All of the following commands assume that you have defined the same environment variables used by your application.

ProTip: To save typing, you can export environment variables for your current bash/terminal session:

```
export DATABASE_URL=postgres://user:pass@host:port/db
```

Create a new migration

To create a new migration file, run the following command and provide a meaningful name:

```
sequelize migration:create --name="meaningful-name"
```

This will create a new file in the migrations folder. All migration file names will be prefixed with a sortable data/time string: `20160421135254-meaningful-name.js`. This prefix is crucial for making sure your migrations are executed in the proper order.

NOTE: The order of your migrations is determined by the alphabetical order of the migration scripts in the file system. The file names generated by the CLI tools will always ensure that the most recent migration comes last.

Add the up/down scripts:

Open the newly created migration file and write the code to both apply and undo the migration. Please refer to the [sequelize migration functions](#) for available operations. **Do not be lazy - write the down script too and test!** Here is an example of converting a `NOT NULL` column accept null values:

```
'use strict';

module.exports = {
  up: function (queryInterface, Sequelize) {
    return queryInterface.changeColumn('tableName', 'columnName', {
      type: Sequelize.STRING,
      allowNull: true
    });
  },
  down: function (queryInterface, Sequelize) {
    return queryInterface.changeColumn('tableName', 'columnName', {
      type: Sequelize.STRING,
      allowNull: false
    });
  }
};
```

ProTip: As of this writing, if you use the `changeColumn` method you must **always** specify the `type`, even if the type is not changing.

ProTip: Down scripts are typically easy to create and should be nearly identical to the up script except with inverted logic and inverse method calls.

Keeping your app code in sync with migrations

The application code should always be up to date with the migrations. This allows the app to be freshly installed with everything up-to-date without running the migration scripts. Your migrations should also never break a freshly installed app. This often times requires that you perform any necessary checks before executing a task. For example, if you update a model to include a new field, your migration should first check to make sure that new field does not exist:

```
'use strict';

module.exports = {
  up: function (queryInterface, Sequelize) {
    return queryInterface.describeTable('tableName').then(attributes => {
      if ( !attributes.columnName ) {
        return queryInterface.addColumn('tableName', 'columnName', {
          type: Sequelize.INTEGER,
          defaultValue: 0
        });
      }
    });
  },
  down: function (queryInterface, Sequelize) {
    return queryInterface.describeTable('tableName').then(attributes => {
      if ( attributes.columnName ) {
        return queryInterface.removeColumn('tableName', 'columnName');
      }
    });
  }
};
```

Apply a migration

The CLI tools will always run your migrations in the correct order and will keep track of which migrations have been applied and which have not. This data is stored in the database under the `_migrations` table. To ensure you are up to date, simply run the following:

```
sequelize db:migrate
```

ProTip: You can add the migrations script to your application startup command to ensure that all migrations have run every time your app is started. Try updating your package.json `scripts` attribute and run `npm start`:

```
scripts: {  
  start: "sequelize db:migrate && node src/"  
}
```

Undo the previous migration

To undo the last migration, run the following command:

```
sequelize db:migrate:undo
```

Continue running the command to undo each migration one at a time - the migrations will be undone in the proper order.

Note: - You shouldn't really have to undo a migration unless you are the one developing a new migration and you want to test that it works. Applications rarely have to revert to a previous state, but when they do you will be glad you took the time to write and test your `down` scripts!

Reverting your app to a previous state

In the unfortunate case where you must revert your app to a previous state, it is important to take your time and plan your method of attack. Every application is different and there is no one-size-fits-all strategy for rewinding an application. However, most applications should be able to follow these steps (order is important):

1. Stop your application (kill the process)
2. Find the last stable version of your app
3. Count the number of migrations which have been added since that version
4. Undo your migrations one at a time until the db is in the correct state
5. Revert your code back to the previous state
6. Start your app

KnexJS



[feathers-knex](#) is a database adapter for [KnexJS](#), an SQL query builder for Postgres, MSSQL, MySQL, MariaDB, SQLite3, and Oracle.

```
npm install --save mysql knex feathers-knex
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

Note: You also need to [install the database driver](#) for the DB you want to use.

API

service(options)

Returns a new service instance initialized with the given options.

```
const knex = require('knex');
const service = require('feathers-knex');

const db = knex({
  client: 'sqlite3',
  connection: {
    filename: './db.sqlite'
  }
});

// Create the schema
db.schema.createTable('messages', table => {
  table.increments('id');
  table.string('text');
});

app.use('/messages', service({
  Model: db,
  name: 'messages'
}));
app.use('/messages', service({ Model, name, id, events, paginate }));


```

Options:

- `Model` (**required**) - The KnexJS database instance
- `name` (**required**) - The name of the table
- `id` (*optional*, default: `'id'`) - The name of the id field property.
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A [pagination object](#) containing a `default` and `max` page size

adapter.createQuery(query)

Returns a KnexJS query with the [common filter criteria](#) (without pagination) applied.

params.knex

When making a [service method](#) call, `params` can contain an `knex` property which allows to modify the options used to run the KnexJS query. See [customizing the query](#) for an example.

Example

Here's a complete example of a Feathers server with a `messages` SQLite service. We are using the [Knex schema builder](#) and [SQLite](#) as the database.

```
$ npm install feathers feathers-errors feathers-rest feathers-socketio body-parser feathers-knex knex sqlite3
```

In `app.js`:

```
const feathers = require('feathers');
const errorHandler = require('feathers-errors/handler');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const service = require('feathers-knex');
const knex = require('knex');

const db = knex({
  client: 'sqlite3',
  connection: {
    filename: './db.sqlite'
  }
});

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }))
  // Create Knex Feathers service with a default page size of 2 items
  // and a maximum size of 4
  .use('/messages', service({
    Model: db,
    name: 'messages',
    paginate: {
      default: 2,
      max: 4
    }
  })
  .use(errorHandler());

// Clean up our data. This is optional and is here
// because of our integration tests
db.schema.dropTableIfExists('messages').then(() => {
  console.log('Dropped messages table');

  // Initialize your table
  return db.schema.createTable('messages', table => {
    console.log('Creating messages table');
    table.increments('id');
    table.string('text');
  });
}).then(() => {
```

```
// Create a dummy Message
app.service('messages').create({
  text: 'Message created on server'
}).then(message => console.log('Created message', message));
});

// Start the server.
const port = 3030;

app.listen(port, () => {
  console.log(`Feathers server listening on port ${port}`);
});
}
```

Run the example with `node app` and go to localhost:3030/messages.

Querying

In addition to the [common querying mechanism](#), this adapter also supports:

\$like

Find all records where the value matches the given string pattern. The following query retrieves all messages that start with `Hello`:

```
app.service('messages').find({
  query: {
    text: {
      $like: 'Hello%'
    }
  }
});
```

Through the REST API:

```
/messages?text[$like]=Hello%
```

Transaction Support

The Knex adapter comes with three hooks that allows to run service method calls in a transaction. They can be used as application wide (`app.hooks.js`) hooks or per service like this:

```
// A common hooks file
const { hooks } = require('feathers-knex');

const { transaction } = hooks;

module.exports = {
  before: {
    all: [ transaction.start() ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
  after: {
```

```

    all: [ transaction.end() ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
  error: {
    all: [ transaction.rollback() ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  }
);

```

To use the transactions feature, you must ensure that the three hooks (start, commit and rollback) are being used.

At the start of any request, a new transaction will be started. All the changes made during the request to the services that are using the `feathers-knex` will use the transaction. At the end of the request, if successful, the changes will be committed. If an error occurs, the changes will be forfeit, all the `creates`, `patches`, `updates` and `deletes` are not going to be committed.

The object that contains `transaction` is stored in the `params.transaction` of each request.

Important: If you call another Knex service within a hook and want to share the transaction you will have to pass `hook.params.transaction` in the parameters of the service call.

Customizing the query

In a `find` call, `params.knex` can be passed a KnexJS query (without pagination) to customize the find results.

Combined with `.createQuery({ query: {...} })`, which returns a new KnexJS query with the [common filter criteria](#) applied, this can be used to create more complex queries. The best way to customize the query is in a `before hook` for `find`.

```

app.service('mesages').hooks({
  before: [
    find(hook) {
      const query = this.createQuery({ query: hook.params.query });

      // do something with query here
      query.orderBy('name', 'desc');

      hook.params.knex = query;
    }
  ]
});

```

RethinkDB



[feathers-rethinkdb](#) is a database adapter for [RethinkDB](#), a real-time database.

```
$ npm install --save rethinkdbdash feathers-rethinkdb
```

Important: To use this adapter you also want to be familiar with the [database adapter common API](#) and [querying mechanism](#).

This adapter requires a running [RethinkDB](#) server.

API

service(options)

Returns a new service instance initialized with the given options. For more information on initializing the driver see the [RehinkDBdash documentation](#).

```
const r = require('rethinkdbdash')({
  db: 'feathers'
});
const service = require('feathers-rethinkdb');

app.use('/messages', service({
  Model: r,
  db: 'someotherdb', //must be on the same connection as rethinkdbdash
  name: 'messages',
  // Enable pagination
  paginate: {
    default: 2,
    max: 4
  }
}));
```

Note: By default, `watch` is set to `true` which means this adapter monitors the database for changes and automatically sends real-time events. This means that, unlike other databases and services, you will also get events if the database is changed directly.

Options:

- `Model` (**required**) - The `rethinkdbdash` instance, already initialized with a configuration object. [see options here](#)
- `name` (**required**) - The name of the table
- `watch` (`options`, default: `true`) - Listen to table changefeeds and send according [real-time events](#) on the adapter.
- `db` (*optional*, default: `none`) - Specify and alternate rethink database for the service to use. Must be on the same server/connection used by rethinkdbdash. It will be auto created if you call `init()` on the service and it does not yet exist.
- `id` (*optional*, default: `'id'`) - The name of the id field property. Needs to be set as the primary key when creating the table.
- `events` (*optional*) - A list of [custom service events](#) sent by this service
- `paginate` (*optional*) - A `pagination object` containing a `default` and `max` page size

adapter.init([options])

Create the database and table if it does not exists. `options` can be the RethinkDB [tableCreate](#) options.

```
// Initialize the `messages` table with `messageId` as the primary key
app.service('messages').init({
  primaryKey: 'messageId'
}).then(() => {
  // Use service here
});
```

adapter.createQuery(query)

Returns a RethinkDB query with the [common filter criteria](#) (without pagination) applied.

params.rethinkdb

When making a [service method](#) call, `params` can contain an `rethinkdb` property which allows to pass additional RethinkDB options. See [customizing the query](#) for an example.

Example

To run the complete RethinkDB example we need to install

```
$ npm install feathers feathers-errors feathers-rest feathers-socketio feathers-rethinkdb rethinkdbdash body-parser
```

We also need access to a RethinkDB server. You can install a local server on your local development machine by downloading one of the packages [from the RethinkDB website](#). It might also be helpful to review their docs on [starting a RethinkDB server](#).

Then add the following into `app.js`:

```
const rethink = require('rethinkdbdash');
const feathers = require('feathers');
const errorHandler = require('feathers-errors/handler');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const service = require('feathers-rethinkdb');

// Connect to a local RethinkDB server.
const r = rethink({
  db: 'feathers'
});

// Create a feathers instance.
var app = feathers()
  // Enable the REST provider for services.
  .configure(rest())
  // Enable the socketio provider for services.
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}))
  // Register the service
  .use('messages', service({
    Model: r,
```

```

    name: 'messages',
    paginate: {
      default: 10,
      max: 50
    }
  ))
  .use(errorHandler());

// Initialize database and messages table if it does not exists yet
app.service('messages').init().then(() => {
  // Create a message on the server
  app.service('messages').create({
    text: 'Message created on server'
  }).then(message => console.log('Created message', message));

  const port = 3030;
  app.listen(port, function() {
    console.log(`Feathers server listening on port ${port}`);
  });
});

```

Run the example with `node app` and go to localhost:3030/messages.

Querying

In addition to the [common querying mechanism](#), this adapter also supports:

\$search

Return all matches for a property using the [RethinkDB match syntax](#).

```

// Find all messages starting with Hello
app.service('messages').find({
  query: {
    text: {
      $search: '^Hello'
    }
  }
});

// Find all messages ending with !
app.service('messages').find({
  query: {
    text: {
      $search: '!$'
    }
  }
});

```

```

GET /messages?text[$search]=^Hello
GET /messages?text[$search]=!$
```

\$contains

Matches if the property is an array that contains the given entry.

```

// Find all messages tagged with `nodejs`
app.service('messages').find({
  query: {
    tags: [

```

```

        $contains: 'nodejs'
    }
});

```

```
GET /messages?tags[$contains]=nodejs
```

Customizing the query

In a `find` call, `params.rethinkdb` can be passed a RethinkDB query (without pagination) to customize the find results.

Combined with `.createQuery(query)`, which returns a new RethinkDB query with the [common filter criteria](#) applied, this can be used to create more complex queries. The best way to customize the query is in a [before hook](#) for `find`. The following example adds a `getNearest` condition for [RethinkDB geospatial queries](#).

```

app.service('mesages').hooks({
  before: {
    find(hook) {
      const query = this.createQuery(hook.params.query);
      const r = this.options.r;

      const point = r.point(-122.422876, 37.777128); // San Francisco

      // Update the query with an additional `getNearest` condition
      hook.params.rethinkdb = query.getNearest(point, { index: 'location' });
    }
  }
});

```

Changefeeds

`.createQuery(query)` can also be used to listen to changefeeds and then send [custom events](#).

Since the service already sends real-time events for all changes the recommended way to listen to changes is with [feathers-reactive](#) however.

Elasticsearch

`feathers-elasticsearch` is a database adapter for [Elasticsearch](#). This adapter is not using any ORM, it is dealing with the database directly through the `elasticsearch.js` [Client](#).

```
$ npm install --save elasticsearch feathers-elasticsearch
```

Getting Started

The following bare-bones example will create a `messages` endpoint and connect to a local `messages` type in the `test` index in your Elasticsearch database:

```
const elasticsearch = require('elasticsearch');
const feathers = require('feathers');
const service = require('feathers-elasticsearch');

app.use('/messages', service({
  Model: new elasticsearch.Client({
    host: 'localhost:9200',
    apiVersion: '5.0'
  }),
  elasticsearch: {
    index: 'test',
    type: 'messages'
  }
}));
```

Options

The following options can be passed when creating a new Elasticsearch service:

- `Model` (**required**) - The Elasticsearch client instance.
- `elasticsearch` (**required**) - Configuration object for elasticsearch requests. The required properties are `index` and `type`. Apart from that you can specify anything that can be passed to all requests going to Elasticsearch. Another recognised property is `refresh` which is set to `false` by default. Anything else use at your own risk.
- `id` (`default: '_id'`) [optional] - The id property of your documents in this service.
- `meta` (`default: '_meta'`) [optional] - The meta property of your documents in this service. The meta field is an object containing elasticsearch specific information, e.g. `_score`, `_type`, `_index`, and so forth.
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the [Pagination chapter](#)).

Complete Example

Here's an example of a Feathers server that uses `feathers-elasticsearch`.

```
const feathers = require('feathers');
const rest = require('feathers-rest');
const hooks = require('feathers-hooks');
const bodyParser = require('body-parser');
const errorHandler = require('feathers-errors/handler');
const service = require('feathers-elasticsearch');
const elasticsearch = require('elasticsearch');
```

```

const messageService = service({
  Model: new elasticseach.Client({
    host: 'localhost:9200',
    apiVersion: '5.0'
  }),
  paginate: {
    default: 10,
    max: 50
  },
  elasticseach: {
    index: 'test',
    type: 'messages'
  }
});

// Initialize the application
const app = feathers()
  .configure(rest())
  .configure(hooks())
  // Needed for parsing bodies (login)
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  // Initialize your feathers plugin
  .use('/messages', messageService)
  .use(errorHandler());

app.listen(3030);

console.log('Feathers app started on 127.0.0.1:3030');

```

You can run this example by using `npm start` and going to localhost:3030/messages. You should see an empty array. That's because you don't have any messages yet but you now have full CRUD for your new message service!

Supported Elasticsearch specific queries

On top of the standard, cross-adapter [queries](#), feathers-elasticsearch also supports Elasticsearch specific queries.

\$all

The simplest query `match_all`. Find all documents.

```

query: {
  $all: true
}

```

\$prefix

Term level query `prefix`. Find all documents which have given field containing terms with a specified prefix (not analyzed).

```

query: {
  user: {
    $prefix: 'bo'
  }
}

```

\$match

Full text query `match`. Find all documents which have given given fields matching the specified value (analysed).

```
query: {
  bio: {
    $match: 'javascript'
  }
}
```

\$phrase

Full text query `match_phrase`. Find all documents which have given given fields matching the specified phrase (analysed).

```
query: {
  bio: {
    $phrase: 'I like JavaScript'
  }
}
```

\$phrase_prefix

Full text query `match_phrase_prefix`. Find all documents which have given given fields matching the specified phrase prefix (analysed).

```
query: {
  bio: {
    $phrase_prefix: 'I like JavaS'
  }
}
```

\$child

Joining query `has_child`. Find all documents which have children matching the query. The `$child` query is essentially a full-blown query of its own. The `$child` query requires `$type` property.

```
query: {
  $child: {
    $type: 'blog_tag',
    tag: 'something'
  }
}
```

\$parent

Joining query `has_parent`. Find all documents which have parent matching the query. The `$parent` query is essentially a full-blown query of its own. The `$parent` query requires `$type` property.

```
query: {
  $parent: {
    $type: 'blog',
    title: {
      $match: 'javascript'
    }
  }
}
```

\$and

This operator does not translate directly to any Elasticsearch query, but it provides support for [Elasticsearch array datatype](#). Find all documents which match all of the given criteria. As any field in Elasticsearch can contain an array, therefore sometimes it is important to match more than one value per field.

```
query: {
  $and: [
    { notes: { $match: 'javascript' } },
    { notes: { $match: 'project' } }
  ]
}
```

There is also a shorthand version of `$and` for equality. For instance:

```
query: {
  $and: [
    { tags: 'javascript' },
    { tags: 'react' }
  ]
}
```

Can be also expressed as:

```
query: {
  tags: ['javascript', 'react']
}
```

\$sqS

[simple_query_string](#). A query that uses the SimpleQueryParser to parse its context. Optional `$operator` which is set to `or` by default but can be set to `and` if required.

```
query: {
  $sqS: {
    $fields: [
      'title^5',
      'description'
    ],
    $query: '+like +javascript',
    $operator: 'and'
  }
}
```

This can also be expressed in an URL as the following:

```
http://localhost:3030/users?$sqS[$fields][]=title^5&$sqS[$fields][]=description&$sqS[$query]=+like +javascript&$sqS[$operator]=and
```

Parent-child relationship

Elasticsearch supports [parent-child relationship](#), however it is not exactly the same as in relational databases. feathers-elasticsearch supports all CRUD operations for Elasticsearch types with parent mapping, and does that with the Elasticsearch constraints. Therefore:

- each operation concerning a single document (create, get, patch, update, remove) is required to provide parent id

- creating documents in bulk (providing a list of documents) is the same as many single document operations, so parent id is required as well
- to avoid any doubts, each query based operation (find, bulk patch, bulk remove) cannot have the parent id

How to specify parent id

Parent id should be provided as part of the data for the create operations (single and bulk):

```
parentService.create({
  _id: 123,
  title: 'JavaScript: The Good Parts'
});

childService.create({
  _id: 1000,
  tag: 'javascript',
  _parent: 123
})
```

Please note, that name of the parent property (`_parent` by default) is configurable through the service options, so that you can set it to whatever suits you.

For all other operations (get, patch, update, remove), the parent id should be provided as part of the query:

```
childService.remove(
  1000,
  { query: { _parent: 123 } }
);
```

Supported Elasticsearch versions

`feathers-elasticsearch` is currently tested on Elasticsearch 2.4, 5.0, 5.1, 5.2, 5.3, 5.4 and 5.5 Please note, even though the lowest version supported is 2.4, that does not mean it wouldn't work fine on anything lower than 2.4.

Quirks

Updating and deleting by query

Elasticsearch is special in many ways. For example, the "["update by query"](#)" API is still considered experimental and so is the "["delete by query"](#)" API introduced in Elasticsearch 5.0.

Just to clarify - update in Elasticsearch is an equivalent to `patch` in feathers. I will use `patch` from now on, to set focus on the feathers side of the fence.

Considering the above, our implementation of path / remove by query uses combo of find and bulk patch / remove, which in turn means for you:

- Standard pagination is taken into account for patching / removing by query, so you have no guarantee that all existing documents matching your query will be patched / removed.
- The operation is a bit slower than it could potentially be, because of the two-step process involved.

Considering, however that elasticsearch is mainly used to dump data in it and search through it, I presume that should not be a great problem.

Search visibility

Please be aware that search visibility of the changes (creates, updates, patches, removals) is going to be delayed due to Elasticsearch `index.refresh_interval` setting. You may force refresh after each operation by setting the service option `elasticsearch.refresh` as described above but it is highly discouraged due to Elasticsearch performance implications.

Full-text search

Currently feathers-elasticsearch supports most important full-text queries in their default form. Elasticsearch search allows additional parameters to be passed to each of those queries for fine-tuning. Those parameters can change behaviour and affect performance of the queries therefore I believe they should not be exposed to the client. I am considering ways of adding them safely to the queries while retaining flexibility.

Performance considerations

None of the data mutating operations in Elasticsearch v2.4 (create, update, patch, remove) returns the full resulting document, therefore I had to resolve to using get as well in order to return complete data. This solution is of course adding a bit of an overhead, although it is also compliant with the standard behaviour expected of a feathers database adapter.

The conceptual solution for that is quite simple. This behaviour will be configurable through a `lean` switch allowing to get rid of those additional gets should they be not needed for your application. This feature will be added soon as well.

Feathers Security

We take security very seriously at Feathers. We welcome any peer review of our 100% open source code to ensure nobody's Feathers app is ever compromised or hacked. As a web application developer you are responsible for any security breaches. We do our very best to make sure Feathers is as secure as possible.

Where should I report security issues?

In order to give the community time to respond and upgrade we strongly urge you report all security issues to us. Send us a PM in [Slack](#) or email us at hello@feathersjs.com with details and we will respond ASAP. Security issues always take precedence over bug fixes and feature work so we'll work with you to come up with a resolution and plan and document the issue on Github in the appropriate repo.

Issuing releases is typically very quick. Once an issue is resolved it is usually released immediately with the appropriate semantic version.

Security Considerations

Here are some things that you should be aware of when writing your app to make sure it is secure.

- Escape any HTML and JavaScript to avoid XSS attacks.
- Escape any SQL (typically done by the SQL library) to avoid SQL injection.
- Events are sent by default to any client listening for that event. Lock down any private events that should not be broadcast by adding [filters](#). Feathers authentication does this for all auth services by default.
- JSON Web Tokens (JWT's) are only signed, they are **not** encrypted. Therefore, the payload can be examined on the client. This is by design. **DO NOT** put anything that should be private in the JWT `payload` unless you encrypt it first.
- Don't use a weak `secret` for your token service. The generator creates a strong one for you automatically. No need to change it.
- Use hooks to check security roles to make sure users can only access data they should be permitted to. We've provided some [built in authorization hooks](#) to make this process easier (many of which are added by default to a generated app).

Some of the technologies we employ

- Password storage inside `feathers-authentication` uses [bcrypt](#). We don't store the salts separately since they are included in the bcrypt hashes.
- [JWT](#) is used instead of cookies to avoid CSRF attacks. We use the `HS512` algorithm by default (HMAC using SHA-512 hash algorithm).
- We run [nsp](#) as part of our CI. This notifies us if we are susceptible to any vulnerabilities that have been reported to the [Node Security Project](#).

XSS Attacks

As with any web application **you** need to guard against XSS attacks. Since Feathers persists the JWT in localstorage in the browser, if your app falls victim to a XSS attack your JWT could be used by an attacker to make malicious requests on your behalf. This is far from ideal. Therefore you need to take extra care in preventing XSS attacks. Our

stance on this particular attack vector is that if you are susceptible to XSS attacks then a compromised JWT is the least of your worries because keystrokes could be logged and attackers can just steal passwords, credit card numbers, or anything else your users type directly.

For more information see:

- [this issue](#)
- and [this Auth0 forum thread](#).

Feathers Ecosystem

Below are some of the amazing things built with Feathers or for the Feathers ecosystem.

We also have a very helpful community in Slack.

slack 31/2069

Production Applications

- [BeachfrontDigital](#)
- [ContactImpact](#)
- [Equibit Group](#)
- [Foxflow](#)
- [GenerousTickets](#)
- [Gratify](#)
- [Headstart](#)
- [HaulHound](#)
- [J.A.B. Property Investments](#)
- [Koola](#)
- [Shakepay](#)
- [Sleeker](#)
- [Simpla](#)
- [Stoplight](#)
- [Taxfyle](#)
- [Work ID](#)

Video tutorials

- [The FeathersJS Youtube playlist](#)
- [FeathersJS Real-Time Chat App - Tutorial](#)
- [Fullstack Feathersjs and React Web App](#)

Starter stacks, Examples and Tutorials

Submit yours by creating a pull request.

- [Feathers Chat](#)
- [Feathers React Native Chat](#)
- [Feathers-Vuex \(Vue.js\) Chat](#)
- [Best Buy API Playground](#)
- [Feathers + Quasar](#)
- [Feathers + Apollo](#)
- [Feathers 2 + Vue 2 + SSR + Email Verification](#)
- [Feathers 2 + Vue 2 + Email Verification + Cordova + Framework 7](#)
- [Feathers + React + Mobx](#)
- [Feathers + React + Webpack](#)

- [Observables with Angular2 and FeathersJS](#)
- [Feathers + React + Redux + Webpack + local auth. Production quality.](#)
- [Live query. Mirror part of a DB on the client.](#)
- [Feathers + React + Redux + Webpack + complete auth + offline mode \(ideal for production\)](#)
- [Build a CRUD App Using React, Redux and FeathersJS](#)
- [feathers-nuxt](#) - A sample/starter for server-side rendered Vue.js + Feathers applications that supports user authentication
- [Passwordless Auth Example Using feathers-authentication-management](#)

Authentication & Authorization

- [feathers-accounts](#) - Token-Based User Account System for FeathersJS (configure).
- [feathers-authentication](#)
- [feathers-authentication-client](#)
- [feathers-authentication-local](#)
- [feathers-authentication-oauth1](#)
- [feathers-authentication-oauth2](#)
- [feathers-authentication-popups](#)
- [feathers-authentication-keystone](#)
- [feathers-permissions](#)
- [feathers-authentication-management](#) - User email verification and password reset capabilities to local feathers-authentication (service)
- [feathers-authentication-compatibility](#) - Keep `v0.x` clients compatible with `v1.0+` authentication

Communications

- [feathers-batch](#) - Batch multiple Feathers service calls into one (service)

Database

- [amity-mongodb](#) - Use various FeatherJS services to manage a MongoDB server with Amity.
- [feathers-blob](#) - Feathers abstract blob store service (service)
- [feathers-blueprints](#) - Add some of the Sails.js blueprints functionality to Feathers. (configure)
- [feathers-bookshelf](#) - A bookshelf ORM service adapter. (service)
- [feathers-couchdb](#)
- [feathers-dynamodb](#) - Work in progress - help wanted!
- [feathers-elasticsearch](#)
- [feathers-filemaker](#) - Filemaker adapter for feathers.js
- [feathers-knex](#)
- [feathers-levelup](#)
- [feathers-linvodb](#) - Create an LinvoDB Service for FeatherJS. (service)
- [feathers-localstorage](#)
- [feathers-memory](#)
- [feathers-mongo-collections](#) - MongoDB collections service for FeathersJS. (service)
- [feathers-mongo-databases](#) - Create a MongoDB database service for FeathersJS. (service)

- [feathers-mongodb](#) ⓘ
- [feathers-mongodb-revisions](#) - This Feathers database adapter extends the basic MongoDB adapter, adding revision support. (service)
- [feathers-mongoose](#) ⓘ
- [feathers-nedb](#) ⓘ
- [feathers-nedb-dump](#) - Middleware for Feathers.js - dumps and restores NeDB database for a given service (middleware)
- [feathers-objection](#) - A service adapter for [Objection.js](#) - A minimal SQL ORM built on top of Knex.
- [feathers-orm-service](#) - Easily create a Object Relational Mapping Service for Featherjs.
- [feathers-rethinkdb](#) ⓘ
- [feathers-rethinky](#) - Thinky.js RethinkDB Adaptor for Feathers JS
- [feathers-seeder](#) - Straightforward data seeder for FeathersJS services.
- [feathers-sequelize](#) ⓘ
- [feathers-skypager](#) - A skypager ORM service adapter (service)
- [feathers-solr](#) - Solr Adapter for Feathersjs
- [feathers-waterline](#)
- [nextql-feathers](#) - Featherjs plugin for NextQL-Yet Another Data Query Language. Equivalent GraphQL but much more simple

Caching

- [feathers-hooks-rediscache](#) - API endpoint caching with Redis.

Documentation

- [feathers-swagger](#) ⓘ - Add documentation to your Feathers services and feed them to Swagger UI. (configure)

Email & SMS

- [feathers-mailer](#) ⓘ - Feathers mailer service using nodemailer (service)
- [feathers-mailgun](#) ⓘ - A Mailgun Service for FeatherJS. (service)
- [feathers-sendgrid](#) ⓘ - A SendGrid Service for FeatherJS. (service)

Microservices

- [mostly-feathers](#) - Convert your Feathers APIs into microservices
- [mostly-feathers-rest](#) - Expose your microservice as a RESTful API

Multiple instances

- [feathers-cluster](#) - Easily take advantage of multi-core systems for Feathers. (configure)
- [feathers-sync](#) ⓘ - Synchronize service events between application instances using MongoDB publish/subscribe (configure)

Mobile

The Feathers client works with React Native but here is a collection of native libraries/SDKs.

- [FeathersjsClientSwift](#) - An iOS client written in Swift.
- [Feathers](#) - Feathers compliant SDK written in Swift 3. Supports rest and socket providers.

Payments

- [feathers-stripe](#) 

Social media

- [feathers-services-instagram-feed](#) - A service that allows to fetch a given user's Instagram feed via its public endpoints.

Testing

- [feathers-tests-fake-app-users](#) - Fake some feathers dependencies in service unit tests. Starter for your customized fakes (service)

Transformation

- [feathers-hooks-csvtoarray](#) - Feathers hook for converting a comma-delimited list to an Array of strings.
- [feathers-hooks-jsonapify](#) - Feathers hook for outputting data in a JSON-API-compliant way.
- [feathers-populate-hook](#) - Feathers hook to populate multiple fields with n:m, n:1 or 1:m relations. (hook)
- [feathers-transform-hook](#) - Feathers hook for transform hook.data parameters (hook)
- [feathers-virtual-attribute-hook](#) - Feathers hook for add virtual attributes to your service response (hook)

Transports

- [feathers-primus](#) 
- [feathers-rest](#) 
- [feathers-socketio](#) 
- [feathers-batch](#) - Batch multiple Feathers service calls into one (service)
- [feathers-socketcluster](#) - Use SocketCluster for client/server communication. Not published.

Utilities

- [feathers-bootstrap](#) 
- [feathers-cli](#) 
- [feathers-client](#) 
- [feathers-commons](#) 
- [feathers-configuration](#) 

- [feathers-errors](#) ⓘ
- [feathers-fs](#) ⓘ - Use the FeathersJS service interface to read and write data in the file system.
- [feathers-generator](#) ⓘ
- [feathers-hooks](#) ⓘ
- [feathers-hooks-common](#) ⓘ - Useful hooks for use with Feathersjs services. (hooks)
- [feathers-hooks-utils](#) - Utility library for writing Feathersjs hooks. (hooks)
- [feathers-logger](#) ⓘ
- [feathers-query-filters](#) ⓘ
- [feathers-profiler](#) ⓘ
- [feathers-socket-commons](#) ⓘ
- [generator-feathers](#) ⓘ
- [generator-feathers-plugin](#) ⓘ
- [feathers-versionate](#) - Utility for creating and working with nested service paths.

Validation

- [feathers-hooks-validate-joi](#) - Feathers hook utility for schema validation, sanitization and client notification using Joi. (hook)
- [feathers-hook-validation-jsonschema](#) - Validate Feathers resources using JSON Schema. (hook)
- [feathers-tcomb](#) - validate feathers services using tcomb (app.service)
- [feathers-validate-hook](#) - Feathers hook for validate json-schema with is-my-json-valid (hook)
- [feathers-validator](#) - A validator for Feathers services. (service)

Client & Framework Integration

- [feathers-client](#) ⓘ - All of the main client packages rolled into one.
- [feathers-mithril](#) - Connect feathers.js to mithril.js (connector)
- [feathers-reactive](#) ⓘ - Turns a Feathers service call into an RxJS observables that automatically updates on real-time events. (configure)
- [feathers-polymer](#)
- [ng-feathers](#) - Feathers client for AngularJS. FeatherJS for plain old AngularJS (1.X)
- [aurelia-feathersjs-socket-demo](#) - Aurelia app (built with Aurelia-CLI) connected to Feathers server application via websockets (socket.io)

DoneJS

- [can-connect-feathers](#) - Feathers client library for DoneJS (feathers-client)
- [canjs-feathers](#) - CanJS model implementation that connects to Feathers services through feathers-client. (feathers-client)
- [donejs-feathers](#) - A generator to quickly add FeathersJS to your DoneJS project. Includes Auth! (generator)

React, Redux

- [feathers-action](#) - use feathers services with redux (connector)

- [feathers-action-creators](#) - redux action creators for feathers services
- [feathers-action-reducer](#) - redux reducer for feathers service actions
- [feathers-action-types](#) - flux action types for feathers services (connector)
- [feathers-react-redux](#) - Unofficial Feathers bindings for React-Redux.
- [feathers-reduxify-services](#) - Wrap Feathers services so they work transparently and perfectly with Redux.
- [feathers-reduxify-authentication](#) - Wrap Feathers.authentication so it works with Redux, and with auth packages for React-Router.

Vue.js

- [feathers-vuex](#) ⓘ - Integration of Feathers services with your Vuex store.
- [vue-syncers-feathers](#) - Synchronises feathers services with vue objects, updated in real time (connector)
- [vue-feathers](#) - A plugin for Vuejs 1.x & 2.x to easily access your feathers services.

Help!

There are many ways that you can get help but before you explore them please check the other parts of these docs, the [FAQ](#), [Stackoverflow](#), [Github Issues](#) and our [Medium publication](#).

If none of those work it's a very real possibility that we screwed something up or it's just not clear. We're sorry . We want to hear about it and are very friendly so feel free to come talk to us in [Slack](#), [submit your issue](#) on Github or ask on [StackOverflow](#) using the [feathersjs](#) tag.

FAQ

We've been collecting some commonly asked questions here. We'll either be updating the guide directly, providing answers here, or both.

How do I create custom methods?

One important thing to know about Feathers is that it only exposes the official [service methods](#) to clients. While you can add and use any service method on the server, it is **not** possible to expose those custom methods to clients.

In the [Why Feathers](#) chapter we discussed how the *uniform interface* of services naturally translates into a REST API and also makes it easy to hook into the execution of known methods and emit events when they return. Adding support for custom methods would add a new level of complexity defining how to describe, expose and secure custom methods. This does not go well with Feathers approach of adding services as a small and well defined concept.

In general, almost anything that may require custom methods can also be done either by creating a [custom service](#) or through [hooks](#). For example, a `userService.resetPassword` method can also be implemented as a password service that resets the password in the `create` method:

```
const crypto = require('crypto');

class PasswordService {
  create(data) {
    const userId = data.user_id;
    const userService = this.app.service('user');

    return userService.patch(userId, {
      passwordToken: crypto.randomBytes(48)
    }).then(user => sendEmail(user))
  }

  setup(app) {
    this.app = app;
  }
}
```

For more examples also see [this issue comment](#).

How do I do nested or custom routes?

Normally we find that they actually aren't needed and that it is much better to keep your routes as flat as possible. For example something like `users/:userId/posts` is - although nice to read for humans - actually not as easy to parse and process as the equivalent `/posts?userId=<userid>` that is already [supported by Feathers out of the box](#). Additionally, this will also work much better when using Feathers through websocket connections which do not have a concept of routes at all.

However, nested routes for services can still be created by registering an existing service on the nested route and mapping the route parameter to a query parameter like this:

```
app.use('/posts', postService);
app.use('/users', userService);

// re-export the posts service on the /users/:userId/posts route
app.use('/users/:userId/posts', app.service('posts'));
```

```
// A hook that updates `data` with the route parameter
function mapUserIdToDate(hook) {
  if(hook.data && hook.params.userId) {
    hook.data.userId = hook.params.userId;
  }
}

// For the new route, map the `:userId` route parameter to the query in a hook
app.service('users/:userId/posts').hooks({
  before: [
    find(hook) {
      hook.params.query.userId = hook.params.userId;
    },
    create: mapUserIdToDate,
    update: mapUserIdToDate,
    patch: mapUserIdToDate
  ]
})
```

Now going to `/users/123/posts` will call `postService.find({ query: { userId: 123 } })` and return all posts for that user.

For more information about URL routing and parameters, refer to [the Express chapter](#).

Note: URLs should never contain actions that change data (like `post/publish` or `post/delete`). This has always been an important part of the HTTP protocol and Feathers enforces this more strictly than most other frameworks. For example to publish a post you would call `.patch(id, { published: true })`.

How do I do search?

This depends on the database adapter you are using. Many databases already support their own search syntax:

- Regular expressions (converted in a hook) for Mongoose, MongoDB and NeDB, see [this comment](#)
- `$like` for [Sequelize](#) which can be set in `params.sequelize`
- Some database adapters like [KnexJS](#), [RethinkDB](#) and [Elasticsearch](#) also support non-standard query parameters which are described in their documentation pages.

For further discussions see [this issue](#).

Why am I not getting JSON errors?

If you get a plain text error and a 500 status code for errors that should return different status codes, make sure you have the `feathers-errors/handler` configured as described in the [Express errors](#) chapter.

Why am I not getting the correct HTTP error code

See the above answer.

How can I do custom methods like `findOrCreate` ?

Custom functionality can almost always be mapped to an existing service method using hooks. For example, `findOrCreate` can be implemented as a before-hook on the service's `get` method. See [this gist](#) for an example of how to implement this in a before-hook.

How do I render templates?

Feathers works just like Express so it's the exact same. We've created a [helpful little guide right here](#).

How do I create channels or rooms

Although Socket.io has a [concept of rooms](#) that you can always fall back to, other websocket libraries that Feathers supports do not. The Feathers way of letting a user listen to e.g. messages on a room is through [event filtering](#). There are two ways:

1. Update the user object with the rooms they are subscribed to and filter based on those

```
// On the client
function joinRoom(roomId) {
  const user = app.get('user');

  return app.service('users').patch(user.id, { rooms: user.rooms.concat(roomId) });
}

// On the server
app.service('messages').filter(function(message, connection) {
  return connection.user.rooms.indexOf(message.room_id) !== -1;
});
```

The advantage of this method is that you can show offline/online users that are subscribed to a room.

1. Create a custom `join` event with a room id and then filter based on it

```
app.use(socketio(function(io) {
  io.on('connection', function(socket) {
    socket.on('join', function(roomId) {
      socket.feathers.rooms.push(roomId);
    });
  });
));

app.service('messages').filter(function(message, connection) {
  return connection.rooms.indexOf(message.room_id) !== -1;
});
```

The room assignment will persist only for the duration of the socket connection.

How do I do validation?

If your database/ORM supports a model or schema (ie. Mongoose or Sequelize) then you have 2 options.

The preferred way

You perform validation at the service level [using hooks](#). This is better because it keeps your app database agnostic so you can easily swap databases without having to change your validations much.

If you write a bunch of small hooks that validate specific things it is easier to test and also slightly more performant because you can exit out of the validation chain early instead of having to go all the way to the point of inserting data into the database to find out if that data is invalid.

If you don't have a model or schema then validating with hooks is currently your only option. If you come up with something different feel free to submit a PR!

The ORM way

With ORM adapters you can perform validation at the model level:

- [Using Mongoose](#)
- [Using Sequelize](#)

The nice thing about the model level validations is Feathers will return the validation errors to the client in a nice consistent format for you.

How do I do associations?

Similar to validation, it depends on if your database/ORM supports models or not.

The preferred way

For any of the feathers database/ORM adapters you can just use [hooks](#) to fetch data from other services.

This is a better approach because it keeps your application database agnostic and service oriented. By referencing the services (using `app.service().find()`, etc.) you can still decouple your app and have these services live on entirely separate machines or use entirely different databases without having to change any of your fetching code.

This has been implemented in the [populate common hook](#).

The ORM way

With mongoose you can use the `$populate` query param to populate nested documents.

```
// Find Hulk Hogan and include all the messages he sent
app.service('user').find({
  query: {
    name: 'hulk@hogan.net',
    $populate: ['sentMessages']
  }
});
```

With Sequelize you can do this:

```
// Find Hulk Hogan and include all the messages he sent
app.service('user').find({
  name: 'hulk@hogan.net',
  sequelize: {
    include: [
      {
        model: Message,
        where: { sender: Sequelize.col('user.id') }
      }
    ]
  }
});
```

Or set it in a hook as [described here](#).

Sequelize models and associations

If you are using the [Sequelize](#) adapter, understanding SQL and Sequelize first is very important. For further information see [this documentation chapter](#) and [this answer on Stackoverflow](#).

What about Koa/Hapi/X?

There are many other Node server frameworks out there like Koa, a *"next generation web framework for Node.JS"* using ES6 generator functions instead of Express middleware or HapiJS etc. Because Feathers 2 is already [universally usable](#) we are planning the ability for it to hook into other frameworks on the server as well. More information can be found in [this issue](#).

How do I filter emitted service events?

See [this section](#).

How do I access the request object in hooks or services?

In short, you shouldn't need to. If you look at the [hooks chapter](#) you'll see all the params that are available on a hook.

If you still need something from the request object (for example, the requesting IP address) you can simply tack it on to the `req.feathers` object [as described here](#).

How do I mount sub apps?

It's pretty much exactly the same as Express. More information can be found [here](#).

How do I do some processing after sending the response to the user?

The hooks workflow allows you to handle these situations quite gracefully. It depends on the promise that you return in your hook. Here's an example of a hook that sends an email, but doesn't wait for a success message.

```
function (hook) {  
  
  // Send an email by calling to the email service.  
  hook.app.service('emails').create({  
    to: 'user@email.com',  
    body: 'You are so great!'  
  });  
  
  // Send a message to some logging service.  
  hook.app.service('logging').create(hook.data);  
  
  // Return a resolved promise to immediately move to the next hook  
  // and not wait for the two previous promises to resolve.  
  return Promise.resolve(hook);  
}
```

How do I debug my app

It's really no different than debugging any other NodeJS app but you can refer to the [Debugging](#) section of the guide for more Feathers specific tips and tricks.

possible EventEmitter memory leak detected warning

This warning is not as bad as it sounds. If you got it from Feathers you most likely registered more than 64 services and/or event listeners on a Socket. If you don't think there are that many services or event listeners you may have a memory leak. Otherwise you can increase the number in the [Socket.io configuration](#) via `io.sockets.setMaxListeners(number)` and with [Primus](#) via `primus.setMaxListeners(number)`. `number` can be `0` for unlimited listeners or any other number of how many listeners you'd expect in the worst case.

Why can't I pass params from the client?

When you make a call like:

```
const params = { foo: 'bar' };
client.service('users').patch(1, { admin: true }, params).then(result => {
  // handle response
});
```

on the client the `hook.params` object will only be available in your client side hooks. It will not be provided to the server. The reason for this is because `hook.params` on the server usually contains information that should be server-side only. This can be database options, whether a request is authenticated, etc. If we passed those directly from the client to the server this would be a big security risk. Only the client side `hook.params.query` and `hook.params.headers` objects are provided to the server.

If you need to pass info from the client to the server that is not part of the query you need to add it to `hook.params.query` on the client side and explicitly pull it out of `hook.params.query` on the server side. This can be achieved like so:

```
// client side
client.hooks({
  before: [
    all: [
      hook => {
        hook.params.query.$client = {
          platform: 'ios',
          version: '1.0'
        };
        return hook;
      }
    ]
  }
});

// server side, inside app.hooks.js
const hooks = require('feathers-hooks-common');

module.exports = {
  before: [
    all: [
      // remove values from hook.params.query.$client and move them to hook.params
      // so hook.params.query.$client.version -> hook.params.version
      // and hook.params.query.$client is removed.
      hooks.client('version', 'platform')
    ]
  ]
};
```

```
}
```

Why are queries with arrays failing?

If you are using REST and queries with larger arrays (more than 21 items to be exact) are failing you are probably running into an issue with the [querystring](#) module which [limits the size of arrays to 21 items by default](#). The recommended solution is to implement a custom query string parser function via `app.set('query parser', parserFunction)` with the `arrayLimit` option set to a higher value. For more information see the [Express application settings feathers-rest#88](#) and [feathers-mongoose#205](#).

I always get a 404 for my custom middleware

Just like in Express itself, the order of middleware matters. If you registered a custom middleware outside of the generator, you have to make sure that it runs before the `NotFound()` error middleware.

How do I get OAuth working across different domains

The standard Feathers OAuth setup sets the JWT in a cookie which can only be passed between the same domain. If your frontend is running on a different domain you will have to use query string redirects as outlined in [this Gist](#).

How do I set up HTTPS?

Check out the Feathers [Express HTTPS docs](#).

Is Feathers production ready?

Yes! It's being used in production by a bunch of companies from startups to fortune 500s. For some more details see [this answer on Quora](#).

Contributing

Just like Feathers itself, all of the documentation is open source and [available to edit on GitHub](#). If you see something that you can contribute, we would LOVE a pull request with your edits! To make this easy you can click the "*Edit this page*" link at the top of the web docs.

The docs are all written in [GitHub Flavored Markdown](#). If you've used GitHub, it's pretty likely you've encountered it before. You can become a pro in a few minutes by reading their [GFM Documentation page](#).

Organizing Files

You'll notice that the [GitHub Repo](#) is organized in a nice logical folder structure. The first file in each chapter is named as a description of the entire chapter's topic. For example, the content related to databases is located in `api/databases/`.

Some of the chapters are split into multiple sections to help break up the content and make it easier to digest. You can easily see how chapters are laid out by looking at the `SUMMARY.md` file. This convention helps keep chapters together in the file system and easy to view either directly on github or gitbook.

Table of Contents

You'll find the table of contents in the `SUMMARY.md` file. It's a nested list of markdown links. You can link to a file simply by putting the filename (including the extension) inside the link target.

Introduction Page

This is the root `README.md` file. Its intent is to give the reader an elevator pitch of what Feathers is and why we think it is useful.

Send a Pull Request

So that's it. You make your edits, keep your files and the Table of Contents organized, and send us a pull request.

Enjoy the Offline Docs

Moments after your edits are merged, they will be automatically published to the web, as a downloadable PDF, .mobi file (Kindle compatible), and ePub file (iBooks compatible).

Share

We take pride in having great documentation and we are very appreciative of any help we can get. Please, let the world know you've contributed to the Feathers Book or give [@FeathersJS](#) a shout out on Twitter to let others know about your changes.

MIT license

Copyright (C) 2017 [Feathers contributors](#)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.